MOMENT: A Formal Framework for MOdel managemMENT

by

Artur Boronat

Submitted to the Department of Information Systems and Computation
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering
at the Universitat Politècnica de València

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

June 2007

Supervisors:
José Meseguer, Isidro Ramos and José Á. Carsí

Table of Contents

List of Tables

List of Figures

Abstract

Model-Driven Development is a field in Software Engineering that, for several years, has represented software artifacts as models in order to improve productivity, quality, and cost effectiveness. Models provide a more abstract description of a software artifact than the final code of the application. In this field, Model-Driven Architecture (MDA) is an initiative, sponsored by the OMG, that is constituted by a family of industry standards including: Meta-Object Facility (MOF), Unified Modeling Language (UML), Object Constraint Language (OCL), XML Metadata Interchange (XMI), and Query/Views/Transformations (QVT). These standards provide common guidelines for model-based tools and processes in order to improve interoperability among executable frameworks, automation in the software development process, and analysis techniques to avoid errors during this process.

The MOF standard describes a generic framework where the abstract syntax of modeling languages can be defined. This standard aims at offering a good basis for Model-Driven Development processes, providing some of the building blocks that are needed to support a Model-Driven Development approach: what is a model, what is a metamodel, what is reflection in a MOF framework, etc. However, most of these concepts lack at present a formal semantics in the current MOF standard. Furthermore, OCL is a constraint definition language that permits adding semantics to a MOF metamodel. Unfortunately, the relation between a metamodel and its OCL constraints also lacks a formal semantics. This is, in part, due to the fact that metamodels can only be defined as data in the MOF framework.

The MOF standard also provides the so-called MOF-Reflection facilities, by means of a generic API, to manipulate software artifacts that are made up out of objects. Broadly speaking, *reflection* is the capacity to represent entities that have a formal semantics at a base level, such as types, as data at a metalevel. Metalevel entities, once metarepresented, can be computationally manipulated and transformed. This notion of *reflection* is still not supported in the MOF standard.

In this dissertation, we define a reflective, algebraic, executable framework for precise metamodeling that provides support for the MOF and the OCL standards. On the one hand, our formal framework provides a formal semantics for the building blocks that are usually involved in a Model-Driven Development process. On the other hand, our framework provides an executable environment that is plugged into the Eclipse Modeling Framework (EMF) and that constitutes the kernel of a model management framework, supporting model transformations and formal verification techniques. The main contributions of this dissertation are:

- The formal definition of the following notions: *metamodel*, *model*, *OCL constraint satisfaction*, and *metamodel conformance*. We clearly distinguish the different roles that the notion of *metamodel* usually plays in the literature: as data, as type and as theory. In addition, we have defined new notions: *metamodel specification*, capturing the relationship between a metamodel and its OCL constraints; *metamodel realization*, referring to the mathematical representation of a metamodel; and *model type* and *constrained model type*, allowing models to be considered as first-class citizens.

- A formal executable mechanism for metamodel reflection in the MOF framework, so that metamodels and models can be manipulated in a generic way.

- An algebraic executable metamodeling framework that is plugged into the Eclipse Modeling Framework, constituting a high-level prototyping environment to experiment with model management tasks where formal verification techniques can be applied.

- A set of prototypes that validate the ideas and notions that have been mathematically defined in this dissertation. These tools provide: OCL constraint validation and OCL query facilities, QVT-based model transformations, and model management tasks by means of generic algebraic operators. Some of these prototypes have already been applied in other universities to explore new areas of research, and in industry. Furthermore, we have shown how our algebraic MOF framework enables the application of formal verification techniques to graph rewriting systems with a simple example.

The outcome of this dissertation constitutes a mathematical executable framework, where formal methods can be applied to several areas of Software Engineering: precise definition of model-based domain-specific languages, model-based formal verification techniques (reachability analysis, model checking, inductive theorem proving, OCL constraint validation, among others), formal definition of model management tasks (model transformation, model merging, traceability support, among others), as well as their application to specific case studies.

Some preliminary experiments related with this dissertation have appeared in [1, 2, 3, 4]. Parts of this work have appeared in [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16].

Keywords: MOF, OCL, Model-Driven Development, Membership Equational Logic, model management, mathematical metamodeling, structural reflection.

Resumen

El Desarrollo de Software Dirigido por Modelos es una rama de la Ingeniería del Software en la que los artefactos software se representan como modelos para incrementar la productividad, calidad y eficiencia económica en el proceso de desarrollo de software, donde un modelo proporciona una representación abstracta del código final de una aplicación. En este campo, la iniciativa Model-Driven Architecture (MDA), patrocinada por la OMG, está constituida por una familia de estándares industriales, entre los que se destacan: Meta-Object Facility (MOF), Unified Modeling Language (UML), Object Constraint Language (OCL), XML Metadata Interchange (XMI), y Query/Views/Transformations (QVT). Estos estándares proporcionan unas directrices comunes para herramientas basadas en modelos y para procesos de desarrollo de software dirigidos por modelos. Su objetivo consiste en mejorar la interoperabilidad entre marcos de trabajo ejecutables, en automatizar el proceso desarrollo de software de software y en proporcionar técnicas que eviten errores durante ese proceso.

El estándar MOF describe un marco de trabajo genérico que permite definir la sintaxis abstracta de lenguajes de modelado. Este estándar persigue la definición de los conceptos básicos que son utilizados en procesos de desarrollo de software dirigidos por modelos: qué es un modelo, qué es un metamodelo, qué es reflexión en un marco de trabajo basado en MOF, etc. Sin embargo, la mayoría de estos conceptos carecen de una semántica formal en la versión actual del estándar MOF. Además, OCL se utiliza como un lenguage de definición de restricciones que permite añadir semántica a un metamodelo MOF. Desafortunadamente, la relación entre un metamodelo y sus restricciones OCL también carece de una semántica formal. Este hecho es debido, en parte, a que los metamodelos sólo pueden ser definidos como dato en un marco de trabajo basado en MOF.

El estándar MOF también proporciona las llamadas facilidades de reflexión de MOF (*MOF Reflection facilities*), mediante una API genérica que permite manipular artefactos software que están compuestos por objetos. De manera informal, *reflexión* es la capacidad para representar entidades que tienen una semántica formal en un nivel base, como por ejemplo tipos, como datos en un metanivel. Las entidades del metanivel pueden ser computacionalmente manipuladas y transformadas. Esta noción de *reflexión* aún no está soportada en el estándar MOF.

En esta tesis, se define un marco de trabajo reflexivo, algebraico y ejecutable para el metamodelado preciso, que proporciona soporte para los estándares MOF y OCL. Por una parte, nuestra aproximación proporciona una definición formal de las nociones básicas que se utilizan en procesos de desarrollo de software dirigidos por modelos. Por otra parte, nuestro marco de trabajo proporciona un entorno ejecutable que está integrado en la herramienta Eclipse Modeling Framework (EMF), y que constituye el núcleo de una herramienta de gestión de modelos, dando soporte a transformaciones de modelos y técnicas de verificación formal. Las principales contribuciones de esta tesis son:

- La definición formal de las siguientes nociones: *metamodelo*, *modelo*, *satisfacción de restricciones OCL*, y *corrección sintáctica* de un modelo. En este trabajo se distingue de una manera clara las diferentes situaciones en las que se utiliza el concepto de metamodelo en la literatura: como dato, como tipo y como teoría. Con este objetivo, se han definido, además, las siguientes nociones: *especificación de metamodelo*, capturando la relación entre un metamodelo y sus restricciones OCL; *realización de un metamodelo*, haciendo referencia a la representación matemática de un metamodelo; y *tipo de modelo* y *tipo de modelo con restricciones*, permitiendo definir modelos como entidades de primer orden.

- Un mecanismo formal y ejecutable de reflexión de metamodelos en el marco de trabajo MOF, de manera que modelos y metamodelos se pueden manipular de manera genérica.
- Un marco de metamodelado algebraico y ejecutable que está integrado en Eclipse Modeling Framework, constituyendo un prototipo de alto nivel para expermientar con tareas de gestión de modelos, en las que se pueden aplicar técnicas de verificación formal.
- Un conjunto de prototipos que validan las ideas y nociones que se han definido matemáticamente en esta tesis. Estas herramientas proporcionan: facilidades para la validación de restricciones OCL y evaluación de consultas OCL, transformaciones de modelos basadas en QVT, y gestión de modelos mediante operadores algebraicos genéricos. Algunos de estos prototipos se han aplicado en otras universidades para explorar nuevas áreas de investigación, y también en industria. Además, se ha presentado cómo se pueden aplicar técnicas de verificación formal a un sistema de reescritura de grafos en nuestro marco de trabajo MOF mediante un ejemplo sencillo.

El resultado de esta tesis constituye un prototipo de metamodelado matemático y ejecutable, donde métodos formales se pueden aplicar a varias áreas de la Ingeniería del Software: definición precisa de lenguajes específicos de dominio basados en modelos, técnicas de verificación formal basadas en modelos (análisis de alcanzabilidad, model checking, demostración inductiva de teoremas, validación de restricciones OCL, entre otras), definición formal de tareas de gestión de modelos (transformación de modelos, integración de modelos, soporte para trazabilidad, entre otros), y sus aplicaciones a casos de estudio específicos.

Algunos experimentos preliminares fueron publicados en [1, 2, 3, 4]. Algunas partes de este trabajo han aparecido en [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16].

Palabras clave: MOF, OCL, Desarrollo de Software Dirigido por Modelos, Lógica Ecuacional de Pertenencia, gestión de modelos, metamodelado matemático, reflexión estructural.

Resum

El Desenvolupament de Programari Dirigit per Models és una branca de l'Enginyeria del Programari, on els artefactes de programari es representen com models per a incrementar la productivitat, qualitat i eficiència econòmica en el procés de desenvolupament de programari, on un model proporciona una representació abstracta del codi final d'una aplicació. En aquest camp, la iniciativa Model-Driven Architecture (MDA), patrocinada per l'OMG, està constituïda per una família d'estàndars industrials, entre els quals destaquem: Meta-Object Facility (MOF), Unified Modeling Language (UML), Object Constraint Language (OCL), XML Metadata Interchange (XMI), i Query/Views/Transformations (QVT). Aquestos estàndars proporcionen unes directrius comunes per a eines basades en models i per a processos de desenvolupament de programari dirigits per models. El seu objectiu consisteix en millorar la interoperabilitat entre marcs de treball executables, en automatitzar el procés de desenvolupament de programri i en proporcionar tècniques que eviten errors durant aquest procés.

L'estàndar MOF descriu un marc de treball genèric que permet definir la sintaxi abstracta de llenguatges de modelat. Aquest estàndar persegueix la definició dels fonaments per a processos de desenvolupament de programari dirigits per models: què és un model, què es un metamodel, què és la reflexió en un marc de treball basat en MOF, etc. No obstant això, la majoria d'aquestos conceptes manquen d'una semàntica formal en la versió actual de l'estàndar MOF. A més a més, OCL s'empra com un llenguatge de definició de restriccions que permet afegir semàntica a un metamodel MOF. Malauradament, la relació entre un metamodel i les seues restriccions OCL també manquen d'una semàntica formal. Aquest fet es deu, en part, a que els metamodels únicament poden ser definits com a data en un marc de treball basat en MOF.

L'estàndar MOF també proporciona les anomenades facilitats de reflexió MOF (*MOF Reflection facilities*), mitjançant una API genèrica que permet manipular artefactes de programari que estan composats per objectes. De manera informal, *reflexió* es la capacitat per a representar entitats que tenen una semàntica formal en un nivell base, com per exemple tipus, com a dades en un metanivell. Les entitats del metanivell poden ser manipulades i transformades computacionalment. Aquesta noció de *reflexió* encara no està suportada en l'estàndar MOF.

En aquesta tesi, es defineix un marc de treball reflexiu, algebraic i executable per al metamodelat precís, proporcionant suport per als estàndars MOF i OCL. D'una banda, la nostra aproximació proporciona una definició formal de les nocions bàsiques que s'empren en processos de desenvolupament de programari dirigits per models. D'altra banda, el nostre marc de treball proporciona un entorn executable que està integrat en l'eina Eclipse Modeling Framework (EMF), i que constitueix el nucli d'una eina de gestió de models, donant suport a transformacions de models i tècniques de verificació formal. Les principals constribucions d'aquesta tesi són:

- La definició formal de les següents nocions: *metamodel*, *model*, *satisfacció de restriccions OCL*, i *correcció sintàctica* d'un model. En aquest treball es distingueix d'una manera clara les diferents situacions en les quals s'empra el concepte de metamodel en la literatura: com a data, com a tipus, i com a teoria. Amb aquest objectiu, s'han incorporat les següent nocions: *especificació de metamodel*, capturant la relació entre un metamodel i les seues restriccions OCL; *realització d'un metamodel*, fent referència a la representació matemàtica d'un metamodel; i *tipus de model* i *tipus de model amb restriccions*, permetent definir models com a entitats de primer ordre.

- Un mecanisme formal i executable de reflexió de metamodels en el marc de treball MOF, de manera que models i metamodels poden ser manipulats de manera genèrica.
- Un marc de metamodelat algebraic i executable que està integrat en Eclipse Modeling Framework, constituint un prototip d'alt nivell per tal d'experimentar amb tasques de gestió de models, en les quals es poden aplicar tècniques de verificació formal.
- Un conjunt de prototips que validen les idees i nocions que s'han definit matemàticament en aquesta tesi. Aquestes eines proporcionen: facilitats per a la validació de restriccions OCL i avaluació de consultes OCL, transformacions de models basades en QVT, i gestió de models mitjançant operadors algebraics genèrics. Alguns d'aquestos prototips s'han aplicat en altres universitats per a explorar noves àrees d'investigació. D'altres s'han aplicat en indústria. A més a més, s'ha presentat com es poden aplicar tècniques de verificació formal a un sistema de reescriptura de grafs en el nostre marc de treball MOF mitjançant un exemple senzill.

El resultat d'aquesta tesi constitueix un prototip de metamodelat matemàtic i executable, on mètodes formals es poden aplicar a diverses àrees de l'Enginyeria del Software: definició formal de llenguatges específics a domini basats en models, tècniques de verificació formal basades en models (anàlisi d'arribada, model checking, demostraci inductiva de teoremes, validació de restriccions OCL, entre altres), definició formal de tasques de gestió de models (transformació de models, integració de models, suport per a traçabilitat, entre altres), i les seues aplicacions a casos d'estudi específics.

Alguns experiments preliminars foren publicats en [1, 2, 3, 4]. Algunes parts d'aquest treball han aparegut en [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16].

Paraules clau: MOF, OCL, Desenvolupament de Programari Dirigit per Models, Lògica Equacional de Pertenència, gestió de models, metamodelat matemàtic, reflexió estructural.

Als maues pares, per l'esforç impagable que han realitzat per tal que pogués realitzar el meu somni.

Als meus germans, Salvador i Carles, que sempre m'han oferit la seua ajuda i estima.

A Fernando i Encarna, els meus nous pares.

A la meua muller, Vicky, per oferir-me la seua estima i rescolzament, fins i tot, a través de la distància que ens ha separat a vegades. Bona part d'aquesta tesi li la dec a ella.

Acknowledgments

During the development of this thesis, I have met many people and I have learnt from all of them. I am grateful to:
Isidro Ramos, because this thesis would not have been finished without his constant support, advice and help. He has not only taught me values for research but also, and more important, for facing life itself.
José Meseguer, who constitutes a source of motivation for me. I would like to thank him his warm welcomes and his patience during my stays in Urbana and, later on, during conference calls...
José Á. Carsí, for trusting me and for his help in the experiences that we have enjoyed during the development of this thesis.
The students, and friends, that have contributed to the development of this thesis, and without whom its impact would not have been the same, specially to: Abel Gómez, Pascual Queralt, Joaquín Oriente, Luis Hoyos, José Iborra, and Julián Pedrós.
CapGemini S.L. España, specially to Pedro J. Molina and José Borrás, for trusting our work and for their support.
I am also very grateful to the members of the ISSI group, to the participants in the DYNAMICA and META projects, and to my friends in UIUC for making my stay a really nice experience, despite the -20 °C. Specially, to Santiago Escobar, for his advices, for his friendship and for such wonderful walks under the *moonlight*.

Introduction

Chapter 1

Introduction

Model-Driven Development is a field in Software Engineering that, for several years, has represented software artifacts as models in order to improve productivity, quality, and cost effectiveness. Models provide a more abstract description of a software artifact than the final code of the application. Roughly speaking, a model can be built by defining concepts and relationships. The set of primitives that permit the definition of these elements constitutes what is called the metamodel of the model.

Interest in this field has grown in software development companies due to several factors. Previous experiences with Model Integrated Computing [17] (where embedded systems are designed and tested by means of models before generating them automatically) have shown that costs decrease in the development process. The consolidation of UML as a design language for software engineers has contributed to Model-Driven Development of software by means of several CASE tools that support the definition of UML models and automated code generation. The emergence of important model-driven initiatives such as the Model-Driven Architecture (MDA) [18, 19], which is supported by OMG, and the Software Factories [20], which is supported by Microsoft, ensures a stock of model-driven technologies for the near future.

Model-Driven Development has evolved into the field of Model-Driven Engineering [21], where not only design and code generation tasks are involved, but also traceability, model management, metamodeling issues, model interchange and persistence, etc. To fulfil these tasks, model transformations and model queries are relevant tasks that must be solved. In the MDA context several open-standards are proposed to handle these tasks. The standard Meta-Object Facility (MOF) standard [22] provides a framework to define metamodels. The standard Query/Views/Transformations (QVT) standard [23] provides support for both transformations and queries. While model transformation technology is being developed, the Object Constraint Language (OCL)[24] remains the best standard choice for queries.

The MOF standard describes a generic framework where the abstract syntax of modeling languages can be defined. This standard aims at offering a good basis for Model-Driven Development processes and tool interoperability providing some of the building blocks that are needed to support a Model-Driven Development approach: what is a model, what is a metamodel, what means that all models that conform to a metamodel satisfy a set of OCL constraints, etc. However, as discussed in Sections 2 and 3, most of these concepts lacks at present an actual semantics in the current MOF standard. Furthermore, OCL is a constraint definition language that permits adding semantics to a MOF metamodel. However, the relation between a metamodel and its OCL constrains lack a formal semantics. This is, in part, due to the fact that metamodels can only be defined as data in the MOF framework.

The MOF standard also provides the so-called MOF-Reflection facilities, by means of a generic API, to manipulate software artifacts that are made up out of objects. Broadly speaking, *reflection* is the capacity to represent entities that have a formal semantics at a base level, such as types, as data at a metalevel. Reflection is a very powerful computational feature, because metalevel entities, once metarepresented, can be computationally manipulated and transformed. Providing actual reflection in the MOF framework requires that the *metamodel* notion should be formally defined. That is, a metamodel should be a mathematical entity, so that it can be represented as data, manipulated by means of the MOF Reflection Facilities, and represented again as a mathematical entity.

In this work, we define a reflective, algebraic, executable framework for precise metamodeling that provides support for the MOF and the OCL standards. On the one hand, our formal framework aims at providing a formal semantics of the building blocks that are usually involved in a Model-Driven Development process. On the other hand, our framework provides an executable environment that is plugged into the Eclipse Modeling Framework (EMF) [25] and that constitutes the kernel of a model management framework, supporting model transformations and formal verification techniques, among other features, as introduced in Section 9. To achieve these goals, we have chosen *Rewriting Logic* (RL)[26] as the underlying formalism. In this work, we have made explicit use, for the most part, of a subset of RL, namely, *Membership Equational Logic* (MEL)[27].

1.1 Structure of the Document

In this work, we describe an algebraic metamodeling framework that provides support for MOF and OCL concepts, and that is integrated with the Eclipse Modeling Framework. The structure of the document is as follows:

Section 2 states the problem under study in this work, i.e., the main notions of a MOF-based metamodeling approach, involving the MOF and OCL standards. In this section, we discuss their meaning and we clearly distinguish the concepts that we study in detail throughout the rest of the work.

Section 3 provides a summary of (sometimes semi-) formal approaches to deal with precise metamodeling and model transformations. In this section, we also describe some other works that has been done in this direction using the same formalism as ours, namely, *Membership Equational Logic* (MEL).

Section 4 provides some preliminary concepts about the underlying formalism *Membership Equational Logic* and their executable representation in the Maude language.

Section 5 provides an overview of the algebraic metamodeling framework, introducing the infrastructure of parameterized MEL theories that constitute the kernel of the framework.

Section 6 provides the formal semantics of the MOF metamodel, an automated reflection mechanism to define the formal semantics of any MOF metamodel, and a reification mechanism to perform the inverse step. In this section, we provide a formal definition of the *structural conformance relation* between a model and its metamodel.

Section 7 provides the algebraic semantics of the OCL language in our framework. In this Section, we formalize the *constrained conformance relation* between a model and a metamodel with OCL constraints.

Section 8 provides the algebraic semantics of the *MOF Reflection Facilities* by formalizing the MOF OBJECT object type, which permits querying and manipulating any object in a model definition, independently of the corresponding object type. This feature gives full formal support for the reflection notion in the MOF framework.

Section 9 provides a brief description of the integration of the algebraic metamodeling framework into the Eclipse Modeling Framework. We also give an example of how graph rewriting can be achieved by means of Rewriting Logic in our algebraic metamodeling framework.

Section 10 gives a comparison of our approach with other related approaches; summarizes the main contributions of our work; and outlines some future work and open research areas.

Part I

Foundations

Chapter 2

Presentation of the Problem

The Meta Object Facility (MOF) [22] provides a metadata management framework and a set of metadata services to enable the development and interoperability of model and metadata-driven systems. Examples of these systems that use MOF include: software modeling and development tools, data warehouse systems, metadata repositories, etc.

In this section, we provide an informal description of the MOF standard by describing the MOF architecture and the main concepts in the MOF metamodel, which are then formally defined in subsequent sections. We also introduce OCL as a constraint definition language that can be used to define well-formedness rules in MOF metamodels, and describe the reflection facilities that are provided in the MOF framework. We also make explicit those aspects of the MOF framework lacking at present a precise mathematical semantics, a topic that we will fully address in Sections 4-8.

2.1 The MOF Modeling Framework

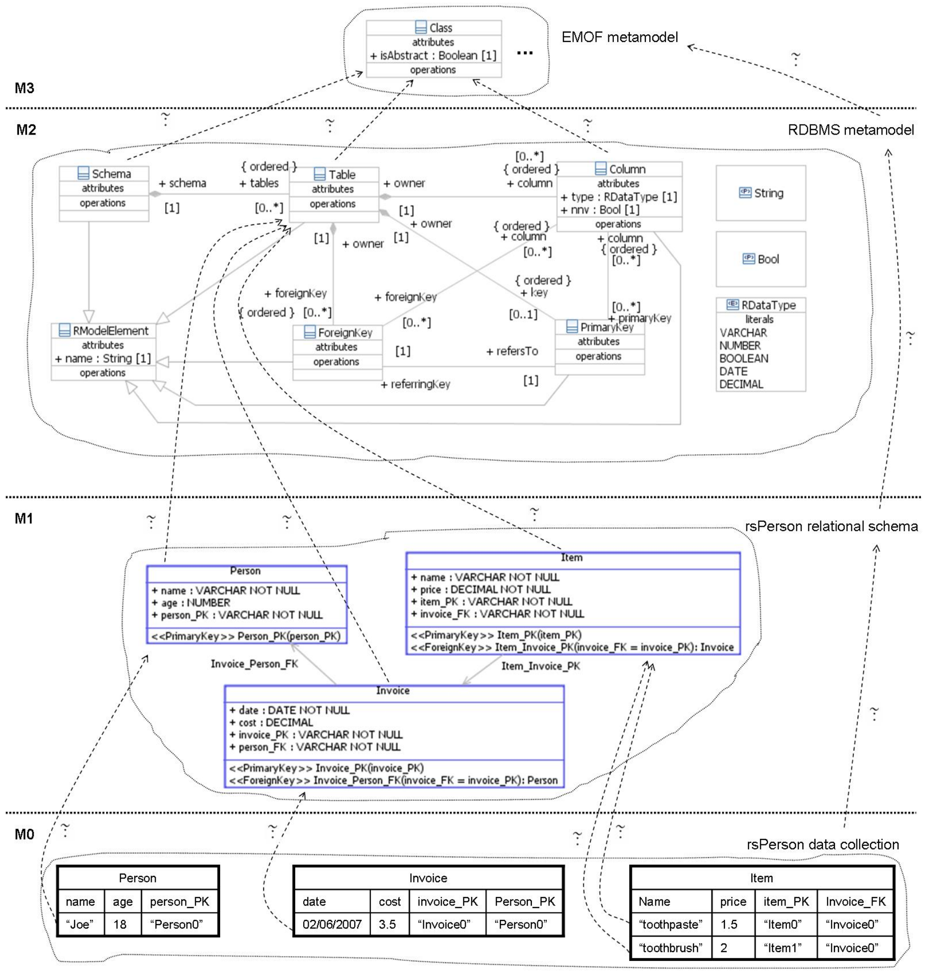
The MOF is a semiformal approach to define modeling languages. It provides a four-level hierarchy, with levels M0, M1, M2 and M3. The entities $\widetilde{m}$ populating each level M_i, written $\widetilde{m} \in M_i$ are always *collections*, made up of constituent *data elements* $\widetilde{e}$. Each entity $\widetilde{M} \in M_{i+1}$ at level i+1 metarepresents a *model*[1] M and is viewed as the metarepresentation of a collection of *types*, i.e., as a metadata collection that defines specific collection of types. Each *type* T is metarepresented as $\widetilde{T} \in \widetilde{M}$ and characterizes a collection of data elements, its *value domain.* We write that a data element $\widetilde{e} \in \widetilde{m}$ is *a value of* type $\widetilde{T} \in \widetilde{M}$ as $\widetilde{e} \mathrel{\tilde{:}} \widetilde{T}$. A metarepresentation at level *i + 1* of a collection $\widetilde{M} \in M_{i+1}$ of types characterizes collections of data elements $\widetilde{m} \in M_i$ at level i. A specific data collection $\widetilde{m} \in M_i$ is said to *conform to* model M, which is metarepresented by its collection of types $\widetilde{M} \in M_{i+1}$, iff for each data element $\widetilde{e} \in \widetilde{m}$ there exists a type $\widetilde{T} \in \widetilde{M}$ such that $\widetilde{e} \mathrel{\tilde{:}} \widetilde{T}$. We write $\widetilde{m} \mathrel{\tilde{:}} \widetilde{M}$ to denote this conformance relation for model M, which we call *structural conformance relation.* The *isValueOf* relation $\widetilde{e} \mathrel{\tilde{:}} \widetilde{T}$ and the *structural conformance relation* $\widetilde{m} \mathrel{\tilde{:}} \widetilde{M}$ are summarized in Fig. 2.1.

$$\begin{array}{ccc} M_i & & M_{i+1} \\ \rotatebox{90}{$\in$} & & \rotatebox{90}{$\in$} \\ \widetilde{m} & \tilde{:} & \widetilde{M} \\ \rotatebox{90}{$\in$} & & \rotatebox{90}{$\in$} \\ \widetilde{e} & \tilde{:} & \widetilde{T} \end{array}$$

Figure 2.1: *isValueOf* and *structural conformance* relations.

Fig. 2.2 illustrates example collections at each level M0-M3 of the MOF framework. Each

[1]In the MOF framework, the concept of a *model M* is conceptually specialized depending on the specific metalevel, in which a model is located: *model* at level M1, *metamodel* at level M2 and *meta-metamodel* at level M3; as shown below.



Person
+ name : VARCHAR NOT NULL
+ age : NUMBER
+ person_PK : VARCHAR NOT NULL
<<PrimaryKey>> Person_PK(person_PK)

Item
+ name : VARCHAR NOT NULL
+ price : DECIMAL NOT NULL
+ item_PK : VARCHAR NOT NULL
+ invoice_FK : VARCHAR NOT NULL
<<PrimaryKey>> Item_PK(item_PK)
<<ForeignKey>> Item_Invoice_PK(invoice_FK = invoice_PK): Invoice

Invoice
+ date : DATE NOT NULL
+ cost : DECIMAL
+ invoice_PK : VARCHAR NOT NULL
+ person_FK : VARCHAR NOT NULL
<<PrimaryKey>> Invoice_PK(invoice_PK)
<<ForeignKey>> Invoice_Person_FK(invoice_FK = invoice_PK): Person

Person

name	age	person_PK
"Joe"	18	"Person0"

Invoice

date	cost	invoice_PK	Person_PK
02/06/2007	3.5	"Invoice0"	"Person0"

Item

Name	price	item_PK	Invoice_FK
"toothpaste"	1.5	"Item0"	"Invoice0"
"toothbrush"	2	"Item1"	"Invoice0"

Figure 2.2: The MOF framework

collection is encircled by a boundary and tagged with a name. This boundary is the non-standard graphical representation of what becomes the physical resource where the model is persisted as data. For example, *rsPerson* $\in M_1$ is a model corresponding to a relational schema. The *isValueOf* relation between elements $\widetilde{e}$ of a data collection and the metarepresentation of types $\widetilde{T}$ of a type collection, and the *structural conformance* relation between a data collection $\widetilde{m}$ and the metarepresentation $\widetilde{M}$ of a model M are depicted with dashed arrows. The four levels M0–M3 in the MOF hierarchy, illustrated in Fig. 2.2 are:

M0 level. In the M0 level, we only consider collections of *data elements* that are manipulated in a running system. For instance, we may have a simple such collection involving a person called "Joe" who is 18 years old and an invoice to Joe for two items for a cost of 3.5 euros.

M1 level. The M1 level contains metarepresentations of *models*. A model is a set of types that describe the elements of some physical, abstract or hypothetical reality by using a well-defined language. A model of a system enhances the communication among system stakeholders during the software development process. In addition, a model is suitable for computer-based interpretation, so that development tasks can be automated. For example, a model can define a relational schema describing the concepts, i.e., types, of *Person*, *Invoice* and *Item*. For example, the type of *Person* is a table *Person*, with columns *name* and *age*; similarly, there is a table *Invoice*, with columns *date* and *cost*; and a table *Item*, with columns *name* and *price*; a foreign key *Invoice_Person_FK*; and a foreign key *Item_Invoice_FK*. Note that our example collection of data elements in M_0 consisting of the person Joe and his invoice and items conforms to this relational schema.

M2 level. The M2 level contains metarepresentations of *metamodels*. A metamodel is a model specifying a modeling language. As an example, we take a simple relational metamodel from the example of the QVT standard that contains the main concepts to define relational schemas, as shown in Fig. 2.2 in UML notation. The types of a relational schema are called *table*, *column*, *foreign key*, etc. Our example model, namely, the relational schema with tables *Person*, *Invoice* and *Item* can be represented as a collection at level M1 that conforms to the relational metamodel at level M2.

M3 level. An entity at the M3 level is the metarepresentation of a *meta-metamodel*. A meta-metamodel specifies a *modeling framework*, which could also be called a *modeling space*. In MOF, there is only one such meta-metamodel, called the MOF meta-metamodel. Within the MOF modeling framework one can define many different metamodels. Such metamodels, when represented as data, must conform to the MOF meta-metamodel. In particular, the relational metamodel conforms to the MOF meta-metamodel. But in MOF one can likewise define many other metamodels, for example the UML metamodel to define UML models, the OWL metamodel to define ontologies, and so on. The fact that all these metamodels are specified within the single MOF framework greatly facilitates systematic model/metamodel interchange and integration.

2.2 Discussion on the Current MOF Standard

At present, important MOF concepts such as those of metamodel, model and conformance relation do not have an explicit, *syntactically characterizable* status in their data versions. For example, we can syntactically characterize the correctness of the data elements in $\widetilde{\mathcal{M}}$ for a metamodel $\mathcal{M}$, but there is no explicit type that permits defining $\widetilde{\mathcal{M}}$ as a well-characterized value. In addition, in the MOF standard and in current MOF-like modeling environments, such as Eclipse Modeling Framework [25], NetBeans MDR [28], DSL tools [29], MetaEdit+ [30], a metamodel $\mathcal{M}$ does not have a precise *mathematical* status. Instead, at best, a metamodel $\mathcal{M}$ is realized as a program in a conventional language, which may be generated from $\widetilde{\mathcal{M}}$, as, for example, the Java code that is generated for a metamodel $\widetilde{\mathcal{M}}$ in EMF. In these modeling environments, the conformance relation between a model definition $\widetilde{M}$ and its corresponding metamodel definition $\widetilde{\mathcal{M}}$ is checked by means of indirect techniques based on XML document validation or on tool-specific implementations in OO programming languages. Therefore, metamodels $\widetilde{\mathcal{M}}$ and models $\widetilde{M}$ cannot be explicitly characterized as first-class entities in their data versions, and the semantics of the conformance relation remains formally unspecified.

In our approach, a formal executable method to define MOF metamodels as precise, mathematical entities is described. Once the mathematical status of a MOF metamodel is made clear, the conformance relation acquires a well-defined algebraic semantics. This mechanism is embodied by the *reflection* concept, which appears partially specified in the MOF standard, as discussed below.

Broadly speaking, *reflection* is the capacity to represent metainformation such as types, which are available at a base level, as *data* in a metadata level. Such form of metarepresentation is usually called *reification*. Reflection is a very powerful computational feature because metalevel entities, once metarepresented, can be computationally manipulated and transformed. After a metadata manipulation of this nature, a mechanism, called *reflection*, permits defining the metarepresented entities of the metalevel as mathematical entities in the base level back again. This concept is illustrated in Fig. 2.3, where $\widetilde{\phi}$ is a metadata level function that permits manipulating metarepresented entities, such as metamodel definitions $\widetilde{\mathcal{M}}$, and where ϕ is the corresponding function at the base level.

In the case of MOF, there are three metalevels, namely M1–M3, structured as a "reflective tower." Each of these metalevels can in turn be split into a base level and a metadata level, as shown in

$$\begin{tikzcd}
\textit{metadata level} & \widetilde{\mathcal{M}} \arrow[r, mapsto, "\widetilde{\phi}"] & \widetilde{\mathcal{M}'} \arrow[d, mapsto, "\textit{reflect}"] \\
\textit{base level} & \mathcal{M} \arrow[u, mapsto, "\textit{reify}"'] \arrow[r, mapsto, "\phi"] & \mathcal{M}'
\end{tikzcd}$$

Figure 2.3: Reflection.

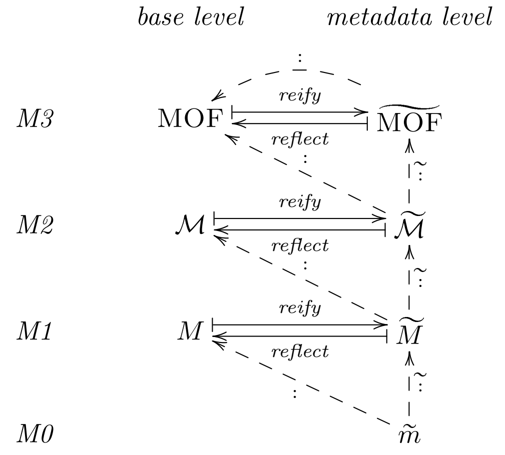


Figure 2.4: The MOF reflective tower

Fig. 2.4. A reification function *reify* : $M \mapsto \widetilde{M}$ maps M in the base sublevel of M_i, $1 \leqslant i \leqslant 3$, to its metarepresentation $\widetilde{M}$ as data in the metadata sublevel of M_i. If $M \in M_1$ is a model, then its reification $\widetilde{M}$ must conform to its metamodel $\mathcal{M} \in M_2$. Now, the structural conformance relation is not metarepresented as data, and we write $\widetilde{M} : \mathcal{M}$. For example, if M is a UML class diagram, $\widetilde{M}$ must conform to the UML metamodel. Similarly, if $\mathcal{M} \in M_2$ is a metamodel, then its reification $\widetilde{\mathcal{M}}$ must conform to the MOF meta-metamodel. What about $\widetilde{MOF}$? The interesting point is that the MOF avoids an infinite reflective tower upwards because its metarepresentation $\widetilde{MOF}$ conforms to the MOF meta-metamodel itself, that is, we have $\widetilde{\text{MOF}}$: MOF. What about the types of $\mathcal{M}$? To answer this question we should consider the inverse process *reify* : $\widetilde{M} \mapsto M$ of passing from a metarepresentation $\widetilde{M}$ to the metalevel entity M that it represents in the corresponding base level. This inverse process is usually called *reflection*.

The MOF standard provides a specification of the so-called MOF Reflection facilities. These focus on the manipulation of the metarepresentation $\widetilde{\mathcal{M}}$ of a MOF metamodel $\mathcal{M}$. The *reflect* and *reify* functions, together with the MOF Reflection facilities, provide full support for metamodel reflection in the MOF framework. Fig. 2.3 illustrates how the metarepresentation of a MOF metamodel $\widetilde{\mathcal{M}}$ can be manipulated by means of a function $\widetilde{\phi} : \widetilde{\mathcal{M}} \mapsto \widetilde{\mathcal{M}'}$ that modifies the objects that define the metamodel $\widetilde{\mathcal{M}}$. Therefore, the function $\phi : \mathcal{M} \mapsto \mathcal{M}'$ that applies the same changes to the mathematical entities of a metamodel can be defined as $\phi = \textit{reflect} \circ \widetilde{\phi} \circ \textit{reify}$.

In its current version, the MOF standard only permits the formal definition of software artifacts in the metadata sublevel of each metalevel (M3-M0). This is due to the lack of a suitable, reflective logic in which the software artifacts, and not just their metarepresentations, can acquire a formal semantics. The current MOF standard does not provide any guidelines to implement the reflect/reify mappings. Only some MOF-like environments provide an informal approach to the *reflect* feature, based on simple code generation. In our approach, we provide a formal definition of the *reflect* function for MOF metamodels. Since our formal specification is executable, this function can also be viewed as a code generation function, but there is a very important conceptual and practical difference: the resulting software artifact can be directly considered as a mathematical theory in Membership Equational Logic, enabling reflective formal reasoning in the MOF framework.

2.3 OCL Constraints in MOF Metamodels

A metamodel definition $\widetilde{\mathcal{M}}$ provides type definitions $\widetilde{T}$ that can be used to define associated data elements $\widetilde{e} \mathrel{\tilde{:}} \widetilde{T}$ in a model definition $\widetilde{M} \mathrel{\tilde{:}} \widetilde{\mathcal{M}}$. Object type definitions $\widetilde{\mathrm{OT}}$ in a metamodel definition $\widetilde{\mathcal{M}}$ constitute subsets $\widetilde{\mathrm{OT}} \subseteq \widetilde{\mathcal{M}}$. A data element that is a value of an object type definition $\widetilde{\mathrm{OT}}$ is called an *object* $\widetilde{o}$. The *isValueOf* relation is refined as the *instanceOf* relation between an object definition $\widetilde{o}$ and the corresponding object type definition $\widetilde{\mathrm{OT}}$. We write $\widetilde{o} \mathrel{\tilde{:}} \widetilde{\mathrm{OT}}$ to denote the metarepresentation of the *instanceOf* relation as data. This relation can also be expressed as $\widetilde{o} : \mathrm{OT}$, when OT is the type that defines the object type, instead of its metadata representation $\widetilde{\mathrm{OT}}$. A model definition $\widetilde{M}$ that conforms to a specific metamodel definition $\widetilde{\mathcal{M}}$, $\widetilde{M} \mathrel{\tilde{:}} \widetilde{\mathcal{M}}$, is a collection of objects $\widetilde{o}$ such that each object is instance of a specific object type definition $\widetilde{\mathrm{OT}} \subseteq \widetilde{\mathcal{M}}$, that is, $\widetilde{o} \mathrel{\tilde{:}} \widetilde{\mathrm{OT}}$.

The Object Constraint Language (OCL) permits defining constraints upon specific object type definitions $\widetilde{\mathrm{OT}}$ in a metamodel definition $\widetilde{\mathcal{M}}$, constraining the objects $\widetilde{o}$ that can be instances of object types $\widetilde{\mathrm{OT}}$ in a model $\widetilde{M} \mathrel{\tilde{:}} \widetilde{\mathcal{M}}$. In subsequent sections, we provide a brief overview of OCL and introduce the mechanism to relate OCL constraints to metamodels. Finally, we introduce the concept of meaningful constraint for a metamodel and the OCL constraint satisfaction relation. The OCL constraint satisfaction relation enriches the structural conformance relation. We call *constrained conformance* relation to the resulting conformance relation.

2.3.1 A Brief Overview of OCL

The Object Constraint Language (OCL) [24] was born as a specification language to complement UML models with constraints and well-formedness requirements, such as invariants, and pre- and post-conditions. The main motivation of the OCL language consists in providing a formal specification language to add expressive power to UML-based models while keeping a textual programming front-end. This makes UML easier to learn for a broad community of system designers and developers. The UML/OCL combination supports both the verification of formal properties in software specifications in the early stages of the software development process and the formal refinement of software specifications into final code. The current version, OCL 2.0, can also be used as a query language for UML-based models. OCL 2.0 is aligned with UML 2.0 and MOF 2.0, so that OCL can also be used in MOF metamodels.

OCL is a strongly typed language without side-effects, where each OCL expression has a type and represents a value, namely, the result of the evaluation of the expression. The evaluation of an expression never changes the system state. For example, if a metamodel is annotated with OCL constraints, these cannot manipulate models that conform to it. The type system of OCL is based on two kinds of types: *predefined types*, which can be decomposed as *basic types* and *collection types*; and *user-defined types*. The predefined basic types are *Integer*, *Real*, *String* and *Bool*. Examples for predefined operations on these types are logical operations like *and*, *or*, *not*, arithmetic operations such as $+$, $-$, $*$, and operations for string manipulation such as *concat*, and *substring*. The predefined collection types are used to specify collections of values. There are four collection types:

- *Set:* A collection of values where order is not relevant and duplicate elements are not allowed.
- *OrderedSet:* A set whose elements are ordered, but not sorted.
- *Bag:* A collection that may contain duplicate elements. Elements in a bag are not ordered.
- *Sequence:* A bag whose elements are ordered, but not sorted.

Among collection operators, we can find universal and existential quantification by means of the *forAll* and *exists* operators, respectively. Collection operators and basic data type operators provide an expressive power close to that of first-order predicate logic together with (finitary) set theory. User-defined types are provided by means of type definitions $\widetilde{T}$ in a metamodel definition $\widetilde{\mathcal{M}}$.

An OCL expression constitutes a *constraint* when the resulting value of its evaluation is a boolean value. When an OCL expression is evaluated, an undefined value may be returned. For example, typecasting of an object to a type that the object does not support, or getting an element from an empty collection. Due to undefined values, OCL constraints can be regarded as sentences in a three-valued logic: *true*, *false*, and *undefined*.

2.3.2 Context of OCL Expressions

OCL expressions rely on type definitions $\widetilde{T}$ of a specific metamodel definition $\widetilde{\mathcal{M}}$. Any metamodel specification $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ in which OCL plays a part consists of a class diagram, the metamodel definition $\widetilde{\mathcal{M}}$, and a set $\widetilde{\mathcal{C}}$ of OCL constraint definitions. The relationship between a type definition $\widetilde{T}$ in a specific metamodel definition $\widetilde{\mathcal{M}}$ and a specific OCL constraint $\widetilde{c}$ is made explicit in the so-called *context* of the OCL constraint $\widetilde{c}$. The type definition $\widetilde{T}$ is then called the *contextual type* of the constraint $\widetilde{c}$ and we denote it as $\mathit{context}(\widetilde{c})$. Usually, an entity $\widetilde{T}$ of the metamodel definition $\widetilde{\mathcal{M}}$ that behaves as context for an OCL constraint is an object type definition[2]. Given a specific metamodel definition $\widetilde{\mathcal{M}}$ and an OCL constraint definition $\widetilde{c}$, we say that $\widetilde{c}$ is a *meaningful OCL constraint* for $\widetilde{\mathcal{M}}$ if its contextual type is a type definition in the metamodel $\widetilde{\mathcal{M}}$, i.e., $\mathit{context}(\widetilde{c}) \in \widetilde{\mathcal{M}}$.

OCL constraints are always evaluated for a single object $\widetilde{o}$, which is always an instance of the corresponding contextual type. In this case, the object $\widetilde{o}$ is called the *contextual instance*. In an OCL constraint definition, the contextual instance can be explicitly referred to by means of the *self* keyword. We write $\widetilde{c}(\widetilde{o})$ to denote the evaluation of the OCL constraint definition $\widetilde{c}$ over the object $\widetilde{o}$.

OCL constraints can be directly incorporated in the class diagram of a metamodel definition, but they may also be provided in a separate text file, where the context definition is given in a textual format. It is then denoted by the `context` keyword followed by the name of the type, as shown in the following example of a context `Foo` for an invariant OCL constraint given by an expression `Bar`:

```
context [var:]  Foo
inv:  Bar
```

where `var:` is a variable of type `Foo`, which is the contextual type of the constraint; the keyword `inv` indicates that the constraint is an invariant that must hold for all the instances of the `Foo` type at any time [3].

In this paper, we have taken as an example the relational metamodel that has been provided in the QVT standard as a case study, shown in Fig. 2.2. In this case, we can use an OCL invariant to ensure that the number of columns that participate in a foreign key must be the same as the number of columns that participate in the corresponding referred primary key. In addition, the type of the columns that participate in a foreign key must be equal to that of each corresponding column (by order) of the referred primary key. This constraint can be expressed in OCL by means of the following invariant:

```
context ForeignKey: inv: if (self.column->size() =
    self.refersTo.column->size()) then
        self.column->forAll(c:Column |
            self.refersTo.column-> at(self.column->indexOf(c)).type
            = c.type
        )
    else
        false
    endif
```

[2]In UML, it can also be an interface, a datatype, or a component. Sometimes it can be an operation, and only rarely it can be an instance.

[3]Constraints of other kinds are also considered in the OCL specification to define pre- and post-conditions for operations. In this paper, we restrict ourselves to the treatment of OCL constraints that are invariants.

where the collection operator `size` computes the cardinality of a collection, the collection operator `forAll` checks if a boolean expression holds over each element of a collection, the collection operator `indexOf` obtains the index of an element in an ordered collection (OrderedSet or Sequence), and the collection operator `at` obtains the element that is located at a given position in an ordered collection. Properties in a class definition are queried by using the "." notation. Collection operators are applied over a collection by means of the `->` construct.

2.3.3 OCL Constraint Satisfaction

An OCL *invariant* $\widetilde{c}$ is a constraint that is defined using a boolean expression that evaluates to *true* if the invariant is met, that is, $\widetilde{c}(\widetilde{o}) : Boolean$. An OCL invariant must hold *true* for any instance of the contextual type at any moment in time. Only when an instance is executing an operation, this does not need to evaluate to *true*. A set of OCL constraints that are meaningful for a metamodel definition $\widetilde{\mathcal{M}}$ may be evaluated over a specific model definition $\widetilde{M} : \mathcal{M}$. More specifically, each OCL constraint definition $\widetilde{c} \in \widetilde{\mathcal{C}}$ is evaluated for each contextual instance $\widetilde{o} \in \widetilde{M}$ such that $\widetilde{o} \mathrel{\tilde{:}} context(\widetilde{c})$. We say that a model $\widetilde{M}$ *satisfies* a set $\widetilde{\mathcal{C}}$ of OCL constraint definitions that are meaningful for a metamodel definition $\widetilde{\mathcal{M}}$ if all such constraints evaluate to *true* for every contextual instance of the model definition $\widetilde{M}$. We write $\widetilde{M} \models \widetilde{\mathcal{C}}$ to denote this OCL constraint satisfaction relation, which is formally expressed as:

$$\widetilde{M} \models \widetilde{\mathcal{C}} \iff$$
$$\forall \widetilde{o} \in \widetilde{M}, \forall \widetilde{c} \in \widetilde{\mathcal{C}}\ ((\widetilde{M} \mathrel{\tilde{:}} \widetilde{\mathcal{M}}\ \wedge\ context(\widetilde{c}) \in \widetilde{\mathcal{M}}\ \wedge\ \widetilde{o} \mathrel{\tilde{:}} context(\widetilde{c})) \rightarrow \widetilde{c}(\widetilde{o}) = true).$$

In the OCL standard, the abstract syntax of the OCL language is provided as a metamodel. Therefore, the metamodel OCL is provided as a model definition $\widetilde{\text{OCL}}$ that conforms to the MOF metamodel, $\widetilde{\text{OCL}} : \text{MOF}$; and a specific OCL constraint c is provided as a model definition $\widetilde{c}$ that conforms to the OCL metamodel, i.e., $\widetilde{c} \mathrel{\tilde{:}} \widetilde{\text{OCL}}$. The OCL standard provides a precise definition of the semantics of both the types of the values that can be used in OCL expressions and the expressions themselves. Some of the types that can be used for values in OCL expressions can be defined by the user in MOF metamodels[4]. The *isValueOf* relation between each value that can be used in an OCL expression and its corresponding type is completely defined in the OCL standard, providing the formal semantics of OCL expressions. Therefore, a set $\widetilde{\mathcal{C}}$ of OCL constraint definitions acquires a mathematical status $\mathcal{C}$, and we can also write $\widetilde{M} \models \mathcal{C}$. At present, [31] provides some guidelines to develop OCL support in modeling environments. However, a rigorous procedure to implement the OCL formal semantics has not been provided, so that the implementation of the standard semantics is left to the developers' programming skills. Thus, the less mathematical the programming language in use is, the more error-prone this task becomes. In this paper, we propose a mechanism to automate this process using executable formal specifications.

The approach that is followed in the standard for the validation of UML models and OCL constraints is based on animation, as formerly described in [32]. A set $\widetilde{\mathcal{C}}$ of OCL constraint definitions, which are meaningful for a metamodel definition $\widetilde{\mathcal{M}}$, can only be checked over specific model definitions $\widetilde{M}$ such that $\widetilde{M} \mathrel{\tilde{:}} \widetilde{\mathcal{M}}$. Therefore, a preliminary notion of model conformance where OCL constraints are taken into account appears between a model definition $\widetilde{M}$ and its metamodel $\mathcal{M}$ (or, analogously, between a snapshot $\widetilde{m}$ and its model M). We call *constrained conformance* relation to the conformance relation that, in addition to requiring structural conformance, also takes the OCL constraint satisfaction relation into account.

However, the provided semantics does not consider either models or snapshots as first-class citizens, so that there is no automated mechanism to formally categorize the set of well-formed models $\widetilde{M}$ that conform to a metamodel definition $\widetilde{\mathcal{M}}$ together with a set $\widetilde{\mathcal{C}}$ of meaningful constraint definitions. Therefore, the constrained conformance relation remains formally unspecified. One of the goals of this paper is to give a precise formal definition of the constrained conformance relation.

[4]We are considering that the formalization that is given for class diagrams in the OCL standard can be directly used for MOF class diagrams.

2.4 Open Problems

In this section we summarize some concepts that are constantly used in Model-Driven Engineering but that lack at present a proper formal semantics. A formal definition of these concepts and their specification in an executable formal framework constitute the main contributions of this paper.

Metamodel Realization. At present, in current MOF-like modeling environments, a metamodel $\mathcal{M}$ does not have a precise *mathematical* status. Instead, at best, a metamodel $\mathcal{M}$ is realized as a program in a conventional language, which may be generated from $\widetilde{\mathcal{M}}$, as, for example, the Java code that is generated for a metamodel $\widetilde{\mathcal{M}}$ in EMF.

Model Type. Metamodels and models are used in Model-Driven Engineering as first-class citizens, but there is no formal, explicit definition of these concepts. Current modeling environments do not provide specific types to define metamodels and models as values, only technologically-based solutions are provided, such as the definition of a model as a XMI document. Therefore, metamodels $\widetilde{\mathcal{M}}$ and models $\widetilde{M}$ cannot be explicitly characterized as first-class citizens in their data versions, and the semantics of the conformance relation remains formally unspecified.

***Structural Conformance* Relation.** In these modeling environments, the conformance relation between a model definition $\widetilde{M}$ and its corresponding metamodel definition $\widetilde{\mathcal{M}}$ is checked by means of indirect techniques based on XML document validation or on tool-specific implementations in OO programming languages. Without considering OCL constraints, a formal characterization of the structural conformance relation is missing. This is, in part, due to the lack of explicit types for metamodels and models.

Metamodel Specification Realization. When a metamodel definition is realized as code in an OO programming language, OCL constraints are usually not considered. They are added afterwards.

Consistent Model Type. Considering OCL constraints in a metamodel definition is still not achieved in an implicit way. Once a model is defined by instantiating the classes of a metamodel in a first step, OCL constraints are validated in a second step. Using metamodels together with their OCL constraints to define model management tasks, such as model transformations, is not straightforward. As indicated in Section 7, another approach consists in defining model types, whose values are model definitions that satisfy the OCL constraints of the corresponding metamodel. In this way, OCL constraints can be taken implicitly when a model is defined. Currently, considering OCL constraints may become a cumbersome task and involves too many technical details in the final solutions.

***OCL Constraint Satisfaction* Relation.** The OCL constraint satisfaction relation is currently defined by means of validation techniques that permit checking if a model definition $\widetilde{M}$ conforms to a metamodel $\mathcal{M}$ together with a set $\mathcal{C}$ of meaningful OCL constraints. However, there is no automated mechanism to formally characterize the set of well-formed models that conform to a metamodel $\mathcal{M}$ that is enriched with OCL constraints $\widetilde{\mathcal{C}}$, that is, conforming to the pair $(\mathcal{M}, \mathcal{C})$. In current MOF formalizations, we cannot implicitly assume that a model definition $\widetilde{M}$ belongs to a certain domain of values of a model type iff it satisfies a set of OCL constraints. This means that a formal characterization of the constrained conformance relation is still missing. Therefore, the application of formal reasoning techniques in a MOF-based framework is still very limited. In our approach the constrained conformance relation is defined in a natural way in the underlying formalism: Membership Equational Logic.

***Reflection* in the MOF Framework.** In the MOF standard, the MOF Reflection facilities are informally described as a single, generic API that permits the manipulation of any MOF metamodel definition $\widetilde{\mathcal{M}}$ or any model definition $\widetilde{M}$ in a type-agnostic way. These facilities provide the basic operations to manipulate model definitions, while OCL remains as a side-effect free language. A complete formal support for reflection also involves reflection and reification mappings, as shown in Fig. 2.3, which permit working not only with the metadata representation $\widetilde{\mathcal{M}}$ of a metamodel, but also with its realization as a mathematical entity, which could be finally refined into a specific application. These mappings are not present in the MOF standard. An informal attempt to realize MOF metamodel definitions as Java programs is provided in the Java Metadata Interface (JMI) specification [33], which is defined for a previous version of the MOF standard. A mapping of this kind has been successfully implemented in modeling environments such as Eclipse Modeling Framework.

Chapter 3

Related Work

3.1 Formal Semantics of Concepts in Model-Driven Development

Model-Driven Engineering is an approach supporting the design of software systems at a conceptual level from where tasks like code generation, interoperability, integration, scalability, quality measurement, among others, can be performed in a mechanical way. [34] summarizes some of the advantages of Model-Driven Engineering. Although industry is becoming more interested in this field, even the most fundamental building blocks of this discipline are still under discussion: starting from informal discussions about the terminology [35] to formal approaches [36, 37] that provide automated support for formal verification, through semi-formal approaches [38, 39, 40, 41, 42]. The common features that characterize these approaches are:

- *Informal approaches:* provide informal definitions of the notions that are used in a model-driven approach for software development. These approaches are usually pioneers and provide new ideas and tools to show proof of concept. However, there is always a gap between the implementation of these tools by means of a specific tool and their underlying mathematical theory (not always existing).

- *Semi-formal discussions:* assume that models are defined, somehow, as subjects in a domain, and model-based notions are studied at a coarser degree. Although all these approaches give more abstract and clearer definitions of the concepts, they are still informal and cannot be proved/checked by means of an objective automated logical system.

- *Formal approaches:* provide a rigorous definition of the notions and the relations that can be used in a model-driven process. These notions are usually defined in a logical theory, where an inference system permits checking the semantics of the notion or of the relations. Therefore, these approaches constitute a more objective alternative to validate the notions. Even in this approach, the theory that defines such notions and relations is developed by a human being. However, the support of an automated logical inference system helps in reducing the degree of informality of the previous approaches and supports formal reasoning and verification.

In subsequent sections, we summarize the notions of *model* and *metamodel* that can be found in the literature.

3.1.1 Model

The discussion on the meaning of the *model* notion is present in the literature, where [43, 38, 40, 39] are just a few citations. There is a consensus on that a model can play several roles:

- *As data:* a model is a collection of structured data elements that syntactically represents some hypothetical or abstract reality, also called the *system under study* (SUS). The definition of the notion of *model* that is given in [38] falls into this category:

  "A model is a set of statements about some SUS."

  where each statement means some predicate about the SUS that can be considered true or false. Seidewitz calls *model interpretation* to the SUS that is denoted by the syntactical representation of a model. Rensink calls *subject* to a model interpretation of this kind, whereas Kühne recalls the notion of *token model* from Peirce's terminology of token/type.

- *As type:* In [39], Rensink characterizes a model as:

  "A subject with one special feature, namely membership test. Essentially the membership test is a function stating, for every subject, whether it is or it is not a member of the model in question."

  This definition reminds us of the notion of type that has a type checking mechanism. This notion of type is developed for model types (not for models as types) in [36, 37]. This membership function is given by the classification function in Kühne's approach, where a model of this kind is called a *type model*. The membership function is further discussed in [41] as Hesse's placeholder projection. These authors assume that the notion of model can play several roles as data or as type, but they do not study their relation. Poernomo provides a formal metamodeling framework based on Constructive Type Theory [36], where models, which are define as terms (token models), can also be represented as types (type models) by means of a reflection mechanism. In this framework, Rensink's membership function is implicitly provided by construction: only valid subjects can be defined as terms, and their definition constitute a formal proof of the fact that the subject belongs to the corresponding type, by means of the Curry-Howard isomorphism.

- *As a specification:* Siedewitz considers the mathematical notion of model that is attached to a term in a theory. In this sense, a theory is a way to deduce new statements about a SUS from the statements that are already given in some model of the SUS. Seidewitz calls model specification to the class of model interpretations that can be build in a theory based on a deductive process. This notion of theory is different from the notion of type. While a *type model* has a membership function associated to it, a model theory constitutes the syntactical representation that contains the syntactical representation of the *type model*. Based on the inference system of the underlying formalism of the model theory, the truth value (soundness) of a *token model* can be checked by means of an automated deductive process.

3.1.2 Metamodel

A metamodel is defined in [38] as

"a specification model for a class of SUS where each SUS in the class is itself a valid model expressed in a valid modeling language,"

where the modeling language is, in fact, defined as data (as a token model) in the form of a metamodel definition. Seidewitz defines the interpretation of a metamodel as a mapping from the elements of a metamodel (as token models) to the elements of the modeling language (theory). This feature constitutes a preliminary notion of reflection that associates the data version of a metamodel to its theory representation. However the notion of type is not developed.

Kühne and Hesse take into account that a model can play the role of type, by means of the classification function. The classification function groups (classifies) values taking into account their types. The notion of *instanceOf* relation is reused in their work to refer to the relation between a token model and its type. However, it is not explained the relation between the *instanceOf* relation for token models and for the objects that constitute a token model, assuming an object-oriented metamodel such as UML.

Kühne defines a metamodel as a model that needs two steps by means of the *instanceOf* relation to represent the system under study. That is, a metamodel permits defining a model type, whose values represent different SUS[1]. Rensink also studies the notion of metamodel as a *language* in a similar way. In his approach, a model is an entity that provides a membership function to determine if a given entity belongs to the model. A *language* is a model that, in addition, induces the membership test for all its members. For example, the Java language allow defining types in programs, which characterize the objects that can be created and manipulated by the program. This approach coincides with Kühne's vision of two meta-layers. Rensink discusses that the MOF meta-metamodel is not a language, because only the data version of a metamodel can be defined. In this case, the semantics of the types that constitute a metamodel remains unspecified.

In Sections 5-8, we describe a reflection mechanism that obtains the semantics for the types that are provided in a metamodel definition $\widetilde{\mathcal{M}}$. This reflection mechanism also takes into account OCL constraints, resulting in an expressive metamodeling framework where the static semantics of metamodels can be defined.

3.2 Formal Metamodeling Approaches

In this section, we study several tools that are used for metamodeling and model transformation purposes. We have focused on tools that are based on formal foundations and that provide formal verification techniques. In this study, we consider the following criteria:

- *Metamodeling approach(two level/multilevel):* As indicated in [44], a multiple metamodeling approach permits dealing with metamodeling frameworks where the number of meta-layers is not fixed. In this approach, both metamodels and models are defined as collections of objects that are related by means of a *instanceOf* relation. This relation is called *ontological instanceOf* relation by Kühne [40]. There is another metamodeling approach based on two levels, namely, the type level and the data level. For example, in a MOF-based approach, a metamodel realization provides the model type $\mathcal{M}$ and the object types that can be used to define objects in a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$. Kühne calls *linguistic instanceOf* relation to the relationship between an object in a model definition and its object type in the corresponding metamodel realization. Despite the fact that the MOF framework is split in four conceptual layers, the linguistic instanceOf relation must be defined for each of the types that participates in a meta-metamodel, in a metamodel or in a model, in levels M3, M2 and M1, respectively.

- *Concrete Syntax:* A metamodeling approach permits defining the semantics of a specific modeling language: usually only the statics and, sometimes, also the dynamics. Some metamodeling approaches also provide facilities to define the concrete syntax of the modeling language: graphical or textual.

- *Static Semantics:* The static semantics of a modeling language is provided by a metamodel definition $\widetilde{\mathcal{M}}$, defining the value domains of the types that are defined in $\widetilde{\mathcal{M}}$, and by a mechanism to check the conformance relation between a model definition $\widetilde{M}$ and a model type $\mathcal{M}$, $\widetilde{M} : \mathcal{M}$. Some metamodeling approaches also allow considering structural constraints, such as OCL constraints, allowing the definition of more expressive metamodels.

- *Dynamic Semantics:* The dynamic semantics of a model permits representing the dynamic behavior of a system, which may be functional or concurrent. A functional system has a

[1]Although, Kühne follows a very generic approach to study ontological metamodeling, where a token model can be considered type of another token model, we restrict ourselves to the so-called linguistic metamodeling, so that we only consider the classification function to define the relation between a token model and its type.

deterministic behavior, while a concurrent system may be non-deterministic. The MOF meta-metamodel lacks suitable constructs to represent the dynamics of a system, whereas the UML metamodel fits better. However, each metamodeling approach provides its own facilities based on its underlying formalism. Some metamodeling approaches that permit representing the non-deterministic semantics of a system in a declarative way also permit defining execution strategies.

- *Reflection:* In a metamodeling framework, *reflection* is the ability to represent a metamodel as data, so that it can be queried or manipulated during a model manipulation task, such as a model transformation. More specifically, introspection refers to the ability for representing a metamodel as data to enable formal reasoning over it. Introspection is a desirable feature in a metamodeling framework that enhances genericity because a model manipulation task can be performed in a way independent of the corresponding metamodel. This feature is discussed in further detail in Section 8.

- *Formal Verification Techniques:* These techniques comprise static analysis techniques to check if a certain property is satisfied during the execution of a non-deterministic system, techniques to study the confluence and termination of the formal definition of a model transformation system, etc.

- *Standards:* The MDA initiative provides a set of standards that are used for metamodeling purposes: MOF, UML, XMI, OCL; and for model transformation purposes: QVT. These standards provide a common conceptual framework to enhance interoperability between different metamodeling approaches.

3.2.1 Formal UML Modeling Environments: The MOVA Framework

UML modeling environments, which may provide support for OCL constraint validation, can also be used for metamodeling purposes by taking into account the UML class diagram and the UML object diagram metamodels. UML modeling environments may also be used for model transformations, although this is not their primary purpose. We have chosen the *MOVA* tool for the comparison study, because it is based on Membership Equational Logic and provides a set of features that are common to our framework. However, the main purposes of the two approaches are different.

The *MOVA* tool [45] is a Maude-based modeling framework that provides support for UML and SecureUML [46] modeling, OCL contraint validation, OCL query evaluation, and OCL-based metrication. *MOVA* supports the UML modeling approach, where metamodel specification definitions $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ can be defined as class diagrams with OCL constraints, and model definitions $\widetilde{M}$ can be defined as object diagrams [47]. Both UML class diagrams and object diagrams are formalized as MEL theories in the *MOVA* Tool. Thus, from and algebraic point of view, the notions of metamodel specification definition and model definition can be syntactically represented as MEL theories, and semantically defined by the corresponding initial algebra [48]. The main purpose of the *MOVA* tool consists in precise modeling, focusing on the static semantics.

Although MEL provides support for reflection, as indicated in Section 4.2, and the implementation of the *MOVA* tool is widely based on this feature, the concept of reflection is still not explicitly available at the UML level. This is in part due to the fact that the UML metamodel does not provide reflection facilities. Therefore, a model can be queried and manipulated as a metarepresented term or as a metarepresented MEL theory in the underlying algebraic representation of the *MOVA* tool, but the notion of an explicit metarepresented model is not considered in the current state of the work.

Since the *MOVA* tool is specified in Maude, it can take advantage of Maude-based formal verification techniques, such as an inductive theorem prover, and tools for checking sufficient completeness, confluence and termination. In addition, operator declarations in a MEL signature can be given in mixfix format so that the concrete syntax of new languages can be taken into account easily.

3.2.2 Graph-based Metamodeling Frameworks

In [49], the authors provide a comparative study between different graph transformation tools. We have based ourselves on [49] to summarize the main features of the tools AGG, ATOM^3 and VIATRA2. However, the criteria that we have proposed above to compare these tools is different, allowing the study of tools that are not graph-based.

AGG

AGG [50] is a development environment for attributed graph transformation systems supporting an algebraic approach to graph transformation. Describing a model transformation by graph transformation, the source and target models have to be given as graphs. Performing model transformation by graph transformation means taking the abstract syntax graph of a model, and transforming it according to certain transformation rules. The result is the abstract syntax graph of the target model.

In this approach, metamodels are defined as type graphs with multiplicities, and models are defined as typed attributed graphs. A class diagram can thus be represented by a type graph plus a set of constraints over this type graph, expressing multiplicities and maybe further constraints. The types that are defined in a type graph can be related by means of subtype relationships [51]. Theoretically, attribute values are defined by separate data nodes, which are elements of some algebra [52]. In *AGG*, attributes are given as Java objects. The Java semantics is not covered by the formal foundation.

A model transformation can be precisely defined by a graph transformation system $GTS = (T, R)$ consisting of a type graph T and a set of transformation rules R. A graph transformation rule $r : L \rightarrow R$ consists of a pair of T-typed graphs L, R such that the union $L \cup R$ is defined. In this case, $L \cup R$ forms a graph again, i.e. the union is compatible with source, target and type settings. The left-hand side L represents the pre-conditions of the rule, while the right-hand side R describes the post-conditions. A rule r may specify attribute computations. The applicability of a rule can be further restricted, if additional application conditions have to be satisfied [53]. Given a host graph and a set of graph rules, two kinds of non-determinism can occur: first several rules might be applicable and one of them is chosen arbitrarily; second, given a certain rule several matches might be possible and one of them has to be chosen. There are techniques to restrict both kinds of choices. Some kind of control flow on rules can be defined by applying them in a certain order or using explicit control constructs, priorities, etc.

Due to its formal foundation, *AGG* offers validation support by means of consistency checking of graphs and graph transformation systems according to graph constraints, critical pair analysis to find conflicts between rules and checking of termination criteria for graph transformation systems. Corresponding criteria are given in [52] for confluence and [54, 55] for termination.

AToM^3

AToM^3 (A Tool for Multi-formalism and Meta-Modeling) [56] is a tool for the design of Domain Specific Visual Languages. It allows defining the abstract and concrete syntax of the Visual Language by means of meta-modeling and expressing model manipulation by means of graph transformation [57]. With the metamodel information, AToM3 generates a customized modeling environment for the described language.

AToM3 permits defining triple graph grammars [58] and multiple views [59]. Triple Graph Grammars [60] were proposed by Andy Schürr as a means to specify translators of data structures, check consistency, or propagate small changes of one data structure as incremental updates into another one. Triple graph grammars can be extended by providing a triple metamodel for typing the triple graphs [58], where metamodels may contain inheritance relations and additional textual constraints. The view support in *AToM*^{3} is very useful when defining multi-view languages, such as UML. In addition, *AToM*^{3} allows the definition of consistency relations between views by means of the aforementioned triple graph grammars.

In *AToM*^{3}, the production rules of a graph grammar can also be defined with application conditions, constraining the context in which they can be applied. In addition, *AToM*^{3} provides a control structure [56], based on rule priorities, for rule execution.

VIATRA2

VIATRA2 [61] is an Eclipse-based general-purpose model transformation framework that provides support for the specification, design, execution, validation and maintenance of transformations within and between various modeling languages and domains.

VIATRA2 uses the VPM metamodeling approach [62] for describing modeling languages and models, which supports arbitrary metalevels in the model space. Queries on models are intuitively captured by generalized (recursive) graph patterns. Model constraints are also captured by the same graph pattern concept, but there is no explicit constraint language for this purpose.

Its rule specification language combines graph transformations and abstract state machines into a single paradigm. Essentially, elementary transformation steps are captured by graph transformation rules (using a recursive graph pattern concept), while complex transformations are assembled from these basic steps by using abstract state machine (ASM) rules as control flow specification. ASMs act as control structures to reduce non-determinism and thus to improve run-time performance.

VIATRA2 provides support for generic and meta-transformations [63] that allow type parameters and manipulate transformations as ordinary models, respectively. This allows arranging common graph algorithms (e.g. transitive closure, graph traversals, etc.) into a reusable library, which is called by assigning concrete types to type parameters in the generic rules. Furthermore, transformations can be externalized by compiling transformations into native Java code, as stand-alone transformation plug-ins. *VIATRA2* transformations may call external Java methods if necessary to integrate external tools into a single tool chain.

3.2.3 Model Checking Graph Transformations

The model checking problem consists in automatically deciding whether a certain correctness property holds in a given system by systematically traversing all enabled transitions in all states (thus all possible execution paths) of the system. The correctness properties are frequently formalized as LTL formulae. The theoretical basics of verifying graph transformation systems by model checking have been studied by Heckel in [64]. The author proposes that graphs can be interpreted as states and rule applications as transitions in a transition system. In [65], two tools that provide support to apply model checking to graph transformations are compared: *CheckVML* and *Groove*. We have used this document to extract a summary of both tools.

The main idea of the *CheckVML* approach [66, 67, 68] consists in exploiting off-the-shelf model checker tools like SPIN [69] for the verification of graph transformation systems. More specifically, it translates a graph transformation system parameterized with a type graph and an initial graph into its Promela equivalent to carry out the formal analysis in SPIN. Furthermore, property graphs are also translated into their temporal logic equivalents, i.e., into the SPIN representation of LTL formulae. *CheckVML* uses directed, typed and attributed graphs to define models. Inheritance between node types in the corresponding type graph is also supported. As indicated in [70], the *CheckVML* tool will be integrated into the *VIATRA2* tool soon.

The idea behind the *GROOVE* approach [71] consists in providing support for model checking, starting from a graph rewriting point of view. This means that states are explicitly represented and stored as graphs, and transitions as applications of graph transformation rules; moreover, properties to be checked should be specified in a graph-based logic, and graph-specific model checking algorithms should be applied. *GROOVE* uses untyped, non-attributed, edge-labeled graphs without parallel edges. It supports the use of negative application conditions.

3.2.4 Discussion: Motivating our Approach

In Table 3.1, we summarize the main features of the aforementioned tools, following the criteria that we have provided above. In the comparative study, we can observe that none of the studied tools provide support for all of the above-mentioned features:

	MOVA	*AGG*	*AToM*^{3}	*VIATRA2*	CheckVML	Groove
metamodeling approach	two levels	two levels	multi-level	multi-level	two levels	–
concrete syntax	mixfix notation*	–	visual	–	–	–
STATICS						
formalism	MEL	attributed typed graphs with inheritance	attributed typed graphs with inheritance	refinement calculus	attributed typed graphs with inheritance	untyped non-attributed labeled graphs
structural constraints	multiplicities, order, uniqueness, OCL	multiplicities application conditions NAC	multiplicities application conditions NAC	recursive pattern matching NAC	NAC	NAC
DYNAMICS	–					
determinism		potentially non-deterministic	potentially non-deterministic	both	potentially non-deterministic	potentially non-deterministic
formalism		graph grammars	graph grammars triple graph grammars	graph grammars + ASM	graph grammars	graph grammars
control structures		rule priorities	rule	ASM priorities		
dynamic constraints		–	–	–	LTL	predicate graph logic
reflection	by means of MEL	–	–	introspection	–	–
formal verification techniques	OCL constraint validation inductive theorem proving	critical pair analysis termination consistency checking	correct typing	correct typing	model checking	model checking
standards	UML,OCL	XML	UML,XML	XML	XML	XML
other features	SecureUML, OCL metrication, OCL queries	graph parsing , attribute values as Java objects	view consistency, bridge to *AGG* tool, attribute values as Python objects, potentially incremental transformations, parallel execution of production rules	generic transformations, meta-transformations, compilation of model transformation, definitions to Java code, interoperability with Java, importer/exporter facilities (EMF)		

Table 3.1: Comparative study.

- Most of the tools provide a *linguistic* approach to deal with types and values, where types are provided by metamodels, and values are defined in model definitions one level down. *VIATRA2* is the tool that follows a multilevel approach.

- The aforementioned metamodeling frameworks are well-suited to define the abstract syntax of a modeling language, but not its concrete syntax. *AToM*^{3} is the most advanced in this sense, providing support for the definition of the visual concrete syntax of a modeling language. The Tiger tool [72] is a metamodeling environment that permits defining graphical environments for metamodels as plugins of the Eclipse platform by using *AGG* as a model transformation engine.

- For defining the static semantics of a modeling language, the OCL language is becoming popular, also in graph-based tools [73, 74]. However, support for OCL is not always available.

- For defining the dynamic semantics of a modeling language, non-standard but formal approaches are used. In the aforementioned tools, graph grammars are the most common choice to define potentially non-deterministic behavior. Sometimes, control structures, like rule priorities or more elaborated constructs, are provided to define a deterministic behavior or to make the execution of a graph transformation more efficient. Model transformations have a functional behavior. This is why some tools, like *AGG*, provide support for determining the confluence and the termination of a graph rewriting system. However, the underlying formalism is not intended for defining functional behavior.

- Reflection is a powerful feature that is used in the *VIATRA2* tool for defining generic model transformations and meta-transformations. The *MOVA* tool can also take advantage of its underlying reflective logic.

- Formal verification techniques are present in all of the chosen approaches. However, each of the tools has been built from scratch pursuing a specific goal, for example, Groove for model checking graph transformations. An approach for precise metamodeling and for model transformation that takes into account a holistic view of all the aforementioned verification techniques seems quite difficult to obtain if we want to build a new tool from scratch.

- Standards are not always taken into account. However, they are a desired feature, since they are intended to enable interoperability between different modeling frameworks. For example, they can be used as interface between precise metamodeling environments and informal development frameworks.

In our approach, we have chosen *Membership Equational Logic* (MEL) [27], introduced in Section 4, as the underlying formalism for our metamodeling framework. Our goal is to provide a formal reflective MOF framework where OCL constraints can be used to define the static semantics of metamodels. In addition, this framework is defined as the basis for a model transformation tool that can benefit from *Rewriting Logic* (RL) [26], a formalism that subsumes MEL. In Section 9, we provide some examples of how our framework can be used to define type graphs and graph instances, and Rewriting Logic is used to define graph grammars. MEL is well-suited for specifying the functional behavior of systems, while RL is ideally suited for specifying concurrent (and possibly non-deterministic) systems. RL, and thereby MEL, are implemented in Maude [75], which provides a flexible parser for user-definable syntax that permits representing context-free grammars as MEL signatures, providing facilities for the definition of the concrete syntax of a language. This is briefly shown in Section 10. The Maude environment also includes an inductive theorem prover, a model checker, and tools for checking sufficient completeness, confluence, and termination of specifications. In addition, Maude provides facilities for debugging, profiling, real-time systems analysis, probabilistic rewriting, reachability analysis, execution strategies, among others (see [75] for a comprehensive overview). Therefore, our motivation consists in formalizing a MOF framework with OCL and reflection facilities, in order to obtain the kernel for a model management framework where Maude-based formal verification techniques can be reused.

3.3 Antecedents

Maude already provides support for object-oriented programming [76], where objects, the *instanceOf* relation and the *class specialization* relation are supported. The dynamics of object-oriented systems can be provided by means of term rewriting.

The static semantics of the UML metamodel (version 1.3) has been previously provided as an algebraic specification in MEL [77]. In this approach, the authors already took the MOF approach into account, although the MOF standard was in its early stages. In [78, 79], the authors provide a formal four-layered framework where: (i) some parts of the MOF meta-metamodel are formalized in a MEL theory at M3 level (called MOF layer); (ii) the UML class diagram and the object diagram metamodels are provided as MEL theories, called *syntactic specification* and *semantic specification* respectively, at M2 level (called UML metamodel layer); (iii) UML class diagrams are defined as terms in the syntactic specification theory at M1 level (called *domain model* layer); and (iv) object diagrams are defined as terms in the semantic specification theory at M0-level (named *user objects* layer). A novel feature in this approach relies on the reuse of the reflective facilities of MEL to provide support for the evolution of UML-based software artifacts [80].

The authors focused on static verification of properties by using Maude as an implementation of MEL and the language to define the constraints. Our work introduces OCL 2.0 as the constraint definition language, the version 2.0 of the MOF standard and considers reflection facilities at a higher-level of abstraction.

Chapter 4

Preliminary Concepts

In this section, we introduce some fundamental concepts of Membership Equational Logic and their representation in Maude. These concepts are used throughout the work and are needed to understand the formal semantics of the notions that are provided in Sections 5-8.

4.1 Membership Equational Logic

A membership equational logic (MEL) [27] *signature* is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$ a many-kinded signature and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. The kind of a sort s is denoted by $[s]$. A MEL Σ-algebra A contains a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \to A_k$ for each operator $f \in \Sigma_{k_1\cdots k_n,k}$ and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$, with the meaning that the elements in sorts are well-defined, while elements without a sort are *errors*. $T_{\Sigma,k}$ and $T_\Sigma(X)_k$ denote, respectively, the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of kinded variables.

Given a MEL signature Σ, *atomic formulae* have either the form $t = t'$ (Σ-equation) or $t : s$ (Σ-membership) with $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$; and Σ-*sentences* are conditional formulae of the form $(\forall X)\ \varphi \ \text{ if } \ \bigwedge_i p_i = q_i \ \wedge \ \bigwedge_j w_j : s_j$, where φ is either a Σ-equation or a Σ-membership, and all the variables in φ, p_i, q_i, and w_j are in X.

A MEL theory is a pair (Σ, E) with Σ a MEL signature and E a set of Σ-sentences. The paper [27] gives a detailed presentation of (Σ, E)-algebras, sound and complete deduction rules, and initial and free algebras. In particular, given an MEL theory (Σ, E), its initial algebra is denoted $T_{\Sigma/E}$; its elements are E-equivalence classes of ground terms in T_Σ.

Order-sorted notation $s_1 < s_2$ can be used to abbreviate the conditional membership $(\forall x : k)\ x : s_2 \ \text{ if } \ x : s_1$. Similarly, an operator declaration $f : s_1 \times \cdots \times s_n \to s$ corresponds to declaring f at the kind level and giving the membership axiom $(\forall x_1 : k_1, \ldots, x_n : k_n)\ f(x_1, \ldots, x_n) : s \ \text{ if } \ \bigwedge_{1\leqslant i\leqslant n} x_i : s_i$. We write $(\forall x_1 : s_1, \ldots, x_n : s_n)\ t = t'$ in place of $(\forall x_1 : k_1, \ldots, x_n : k_n)\ t = t' \ \text{ if } \ \bigwedge_{1\leqslant i\leqslant n} x_i : s_i$.

We can use order-sorted notation as syntactic sugar to present a MEL theory (Σ, E) in a more readable form as a tuple $(S, <, \Sigma, E_0 \cup A)$ where: (i) S is the set of sorts; (ii) $<$ is the subsort inclusions, so that there is an implicit kind associated to each connected component in the poset of sorts $(S, <)$; (iii) Σ is given as an order-sorted signature, that is, giving different (possibly overloaded) operator declarations; and (iv) the set E of (possibly conditional) equations and memberships is quantified with variables having specific sorts (instead than with variables having specific kinds) in the sugared fashion described above; furthermore, E is decomposed as a disjoint union $E = E_0 \cup A$, where A is a collection of "structural" axioms such as associativity, commutativity, and identity. As explained above, any theory $(S, <, \Sigma, E_0 \cup A)$ can then be desugared into a standard MEL an theory (Σ, E).

The point of the decomposition $E = E_0 \cup A$ is that, under appropriate executability requirements explained in [81], such as confluence, termination, and sort-decreasingness modulo A, an MEL theory $(S, <, \Sigma, E_0 \cup A)$ becomes *executable* by rewriting with the equations and memberships E_0 *modulo* the structural axioms A. Furthermore, the initial algebra $T_{\Sigma/E}$ then becomes isomorphic to the *canonical term algebra* $Can_{\Sigma/E_0,A}$ whose elements are A-equivalence classes of ground Σ-terms that cannot be further simplified by the equations and memberships in E_0.

4.2 Reflection

Reflection is a very important property of membership equational logic and of rewriting logic [82]. Intuitively, a logic is reflective if it can represent its metalevel at the object level in a sound and coherent way. Specifically, membership logic can represent its own theories and their deductions by having a finitely presented MEL theory $\mathcal{U}$ that is *universal*, in the sense that for any finitely presented MEL theory $T = (\Sigma, E)$ (including $\mathcal{U}$ itself) and for each Σ-equation or Σ-membership ϕ we have the following equivalence

$$T \vdash \phi \;\Leftrightarrow\; \mathcal{U} \vdash \overline{T \vdash \phi},$$

where $\overline{T \vdash \phi}$ is the sentence in $\mathcal{U}$ stating at the metalevel that ϕ is provable in T. Since $\mathcal{U}$ is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection [83, 84].

Reflection is a very powerful property: it allows powerful meta-programming uses. It is used extensively in our implementation of the *reflect* function, which assigns to each metamodel specification in MOF a corresponding MEL theory as its algebraic semantics. Although for the most part we describe the *reflect* function at the object level; that is, by explaining precisely which MEL theory T is associated by *reflect* to each metamodel specification in MOF, our actual partial implementation in Maude of *reflect* does not map a metamodel specification to T itself, but to the *term* $\overline{T}$ which meta-represents T as data in the universal theory $\mathcal{U}$.

As we further explain in what follow, functional modules in the Maude language are exactly MEL theories. In particular, reflection is a key MEL feature supported by Maude (also for rewriting logic, which extends MEL). Specifically, reflection is efficiently supported in Maude through its `META-LEVEL` module, which provides efficient built-in support in the form of *descent functions* [84] for key functionality in the universal theory $\mathcal{U}$. In particular, `META-LEVEL` has two key sorts: (i) the `Module` sort, whose elements are the meta-representations $\overline{T}$ of theories T; and (ii) the `Term` sort, whose elements are the meta-representations $\bar{t}$ of terms t belonging to some theory T.

4.3 Maude

In this Section, we provide a brief description of the syntactical constructs of the Maude language that we are using throughout the document. For a detailed explanation of the Maude language and its semantics, we refer to [75], which we have used as a basis for this summary.

Maude is a declarative language in the strict sense of the word. That is, a Maude program is a *logical theory*, and a Maude computation is *logical deduction* using the axioms specified in the theory/program. In Maude, the basic units of specification and programming are called *modules*. A module consists of syntax declarations, providing an appropriate language to describe the system at hand, and of statements, asserting the properties of such a system. Membership equational theories are specified as *functional modules* in Maude, where the statements are given in the form of *equations* and *memberships*.

From a programming point of view, a *functional module* is an equational-style functional program with user-definable syntax in which a number of sorts, their elements, and functions on those sorts are defined. Computation is of course the efficient form of equational deduction in which equations are used from left to right as simplification rules. From a specification viewpoint, a functional module is an *equational theory* (Σ, E) with initial algebra semantics. The syntax declaration part is called a signature and consists of declarations for:

- *Sorts:* giving names for the types of data. A sort is declared using the `sort` keyword followed by an identifier (the sort name), followed by white space and a period, as follows:

  `sort` ⟨ *Sort* ⟩ .

- *Subsorts:* organizing the data types in a hierarchy. Subsort inclusions are declared using the keyword subsort. The declaration

  `subsort ⟨ Sort-1 ⟩ < ⟨ Sort-2 ⟩ .`

  states that the first sort is a subsort of the second.

- *Kinds:* which are implicit and intuitively correspond to error supertypes that, in addition to normal data, can contain error expressions. This notion is used in our framework to provide support for undefined values in OCL expressions in the MOF framework. In Maude modules, kinds are not independently and explicitly named. Instead, a kind is identified with its equivalence class of sorts and can be named by enclosing the name of one or more of these sorts in square brackets `[...]`; when using more than one sort, sorts are separated by commas.

- *Operators:* providing names for the operations that will act upon the data and allowing us to build expressions (or terms) referring to such data. In a Maude module, an operator is declared with the keyword `op` followed by its *name*, followed by a colon, followed by the list of sorts for its arguments (called the operators *arity* or *domain sorts*), followed by `->`, followed by the sort of its result (called the operators *coarity* or *range sort*), optionally followed by an attribute declaration (where attributes to indicate associativity, commutativity and identity, among others, can be used [1]), followed by white space and a period. Thus the general scheme has the form

  `op ⟨OpName⟩ : ⟨Sort-1⟩ ... ⟨Sort-k⟩ -> ⟨Sort⟩ [ ⟨OperatorAttributes⟩ ] .`

  To emphasize the fact that an operator defined at the kind level in general defines only a *partial* function at the sort level, Maude also supports a notational variant in which an (always total) operator at the kind level can equivalently be defined as a partial operator between sorts in the corresponding kinds, with syntax '`~>`' instead of '`->`' to indicate partiality.

For example, the following module specifies the definition of natural numbers:

```
fmod NAT is
  sorts Zero NzNat Nat .
  subsorts Zero NzNat < Nat .
  op 0 : -> Zero .
  op s_ : Nat -> NzNat .
endfm
```

where `s` represents the successor function. Using the `NAT` theory, specified as the `NAT` functional module, the natural number 2 can be defined as `s(s(0))`.

The semantics of the operators of a logical signature is defined by means of *equations* and *memberships*. Unconditional equations are declared using the keyword `eq`, followed by a term (its lefthand side), the equality sign `=`, then a term (its right hand side), optionally followed by a list of statement attributes enclosed in square brackets, and ending with white space and a period:

`eq ⟨Term-1⟩ = ⟨Term-2⟩ [ ⟨StatementAttributes⟩ ] .`

Unconditional membership axioms specify terms as having a given sort. They are declared with the keyword `mb` followed by a term, followed by '`:`', followed by a sort (that must always be in the same kind as that of the term), followed by a period. As equations, memberships can optionally have statement attributes:

`mb ⟨Term⟩ : ⟨Sort⟩ [ ⟨StatementAttributes⟩ ] .`

[1]see [75] for a more detailed presentation of these attributes.

Equational conditions in conditional equations and memberships are made up of individual equations $t = t'$ and memberships $t : s$. A condition can be either a single equation, a single membership, or a conjunction of equations and memberships using the binary conjunction connective /\ which is assumed to be associative. Thus the general form of conditional equations and conditional memberships is the following:

```
ceq ⟨Term-1⟩ = ⟨Term-2⟩
if EqCondition-1 /\ ... /\ EqCondition-k [ ⟨StatementAttributes⟩ ] .
```

```
cmb ⟨Term⟩ : ⟨Sort⟩
if EqCondition-1 /\ ... /\ EqCondition-k [ ⟨StatementAttributes⟩ ] .
```

Furthermore, the concrete syntax of equations in conditions has three variants, namely:

- ordinary equations `t = t'`,
- matching equations `t := t'`, and
- abbreviated Boolean equations of the form `t`, with `t` a term in the kind `[Bool]`, abbreviating the equation `t = true`.

The terms `t` and `t'` in an equation `t = t'` must both have the same kind. A further feature, greatly extending the expressive power for specifying partial functions, is the possibility of defining sorts by means of equational conditions. Our MOF framework is widely based on this feature to define the semantics of model types in Section 6 and constrained model types in Section 7.

4.3.1 Parameterized programming

Theories, parameterized modules, and views are the basic building blocks of parameterized programming [85, 86]. Parameterized programming is widely used in the formal specification of our MOF framework to define the generic semantics of metamodels and OCL constructs.

A *parameterized module* is a module with one or more *parameters*, each of which is expressed by means of one theory, that is, modules can be parameterized by one or more theories. If we want, e.g., to define a list or a set of elements, we may define a module `LIST` or `SET` parameterized by a theory expressing the requirements on the type of the elements to store in such data structures. Thus, theories are used to declare the interface requirements for parameterized modules.

Theories are used to declare module interfaces, namely the syntactic and semantic properties to be satisfied by the actual parameter modules used in an instantiation. As for modules, Maude supports two different types of theories: functional theories and system theories, with the same structure of their module counterparts, but with a different semantics. Functional theories are declared with the keywords `fth ... endfth`, and system theories with the keywords `th ... endth`. Both of them can have sorts, subsort relationships, operators, variables, membership axioms, and equations, and can import other theories or modules. The theory TRIV is used very often, for instance in the definition of data structures, such as lists, sets, trees, etc., which are made out of basic elements of some type with no specific requirement. To express this simple requirement, namely a parameter type with no additional requirements we use the theory TRIV, which is predefined in Maude as follows:

```
fth TRIV is
  sort Elt .
endfth
```

The instantiation of the formal parameters of a parameterized module with actual parameter modules or theories requires a *view,* mapping entities from the formal interface theory to the corresponding entities in the actual parameter module. In the definition of a view we have to indicate its name, the source theory, the target module or theory, and the mapping of each sort and operator in the source theory. The name space of views is separate from the name space of modules and theories, which means that, e.g., a view and a module could have the same name. The syntax for views is as follows:

```
view ⟨ViewName⟩ from ⟨Source⟩ to ⟨Target⟩ is
  ⟨Mappings⟩
endv
```

The mapping of a sort in the source theory to a sort in the target module or theory is expressed with syntax

```
sort ⟨identifier⟩ to ⟨identifier⟩ .
```

For each sort S in the source theory, there must exist a sort S' in the target module or theory which is its mapping under the view; unmentioned sorts get the identity mapping. Operators can also be mapped in a view, see [75] for a detailed explanation. For example to map the `TRIV` theory to the `NAT` functional module, we can use the following view:

```
view Nat from TRIV to NAT is
  sort Elt to Nat .
endview
```

We can also have views between theories, which is particularly useful to compose instantiations of views. A view between theories links the formal parameter of some parameterized module to some actual parameter via some intermediate formal parameter of another parameterized module.

System modules and functional modules can be parameterized. A parameterized functional module has syntax

`fmod M{`X_1` :: `T_1` , ... , `X_n` :: `T_n`} is ... endfm`

with $n \geqslant 1$. Parameterized system modules have completely analogous syntax. The $\{X_1$ :: T_1 , ... , X_n :: $T_n\}$ part is called the *interface*, where in each pair X_i :: T_i, X_i is an identifier—the *parameter name* or *parameter label*—and each T_i is an expression that yields a theory—the *parameter theory.*

For example, we define a parameterized functional module that specifies parameterized sets. This module will be further develop in Section 6 as the `OCL-COLLECTION-TYPES{T :: TRIV}` theory:

```
fmod SET{X :: TRIV} is
  sorts Magma{X} NeSet{X} Set{X} .
  subsort NeSet{X} < Set{X} .
  subsort X$Elt < Magma{X} .
  op _,_ : Magma{X} Magma{X} -> Magma{X} [assoc comm] .
  op Set{_} : Magma{X} -> NeSet{X} .
  op empty-set : -> Set{X} .
endfm
```

In this module, the terms of sort `Magma{X}` represent sets of elements, where the elements are separated by commas. The `Set{_}` operator allows defining sets by using the concrete syntax of the OCL language. While `Set` is the sort for sets that may be empty, `NeSet` is the sort for sets that are not empty. The `X$Elt` sort refers to the `Elt` sort that is defined in the parameter theory `TRIV`. Terms of sort `X$Elt` can participate as elements in a set.

Instantiation is the process by which actual parameters are bound to the formal parameters of a parameterized module or theory and a new module is created as a result. The instantiation requires a view from each formal parameter to its corresponding actual parameter. Each such view is then used to bind the names of sorts, operators, etc. in the formal parameters to the corresponding sorts, operators (or expressions), etc. in the actual target. For example, the `SET{X :: TRIV}` module can be instantiated with the `NAT` theory by means of the expression `SET{Nat}`, that is, by using the view `Nat`. In the `SET{Nat}` module we can define sets of natural numbers as follows: `Set{s(0), 0, s(s(0))}`[2].

[2]Maude also supports representing natural numbers in normal decimal notation if we use its `NAT` module. However, we have used a simpler version of this module in the example.

Part II

A Formal MOF Framework

Chapter 5

A High-Level View of the MOF Algebraic Semantics

The practical usefulness of a formal semantics for a language is that it provides a rigorous standard that can be used to judge the correctness of an implementation. For example, if a programming language lacks a formal semantics, compiler writers may interpret the informal semantics of the language in different ways, resulting in inconsistent and diverging implementations. For MOF, given its genericity, the need for a formal semantics that can serve as a rigorous standard for any implementation is even more pressing, since many different modeling languages rely on the correctness of the MOF infrastructure. There is, furthermore, another reason for the usefulness of a formal semantics for metamodeling frameworks, namely, that at present it is unclear whether the differences between various metamodeling frameworks are merely *verbal*, or there exist substantial *semantic* differences. A formal semantics may greatly help in cutting through the confusions caused by merely verbal differences and can make explicit the really important, semantic ones.

In this section, we propose an algebraic, mathematical semantics for MOF in Membership Equational Logic (MEL). However, to better motivate our semantics, we first present an *informal* semantics for MOF that our formal semantics will later make entirely precise and rigorous. At the end of this section, we introduce the structure that is used in Sections 6-8 to define the MOF formal semantics in detail.

5.1 Informal Semantics of MOF

The MOF semantics we present is *informal* because, as already mentioned, at present, a metamodel $\mathcal{M}$ does not have a precise mathematical status, except for other formalization proposals such as [87]. Similarly, the *conforms to* relation also lacks a precise mathematical status. Finally, except for some recent proposals such as [47, 88], in MOF-compliant modeling environments, the satisfaction of a set $\mathcal{C}$ of OCL constraints by a model $\widetilde{M}$ is either checked by conventional code, instead than by a deductive process, or is not checked at all.

Our approach here is to *pretend* that: (i) metamodels, (ii) the structural conformance relation, (iii) metamodel realizations, (iv) metamodel specifications, (v) the OCL constraint satisfaction relation, (vi) the constrained conformance relation, and (vii) metamodel specification realizations, already have a precise mathematical meaning, and to use set-theoretic notation to make explicit the corresponding informal semantics. However, the semantics itself remains, for the moment, informal, despite the set-theoretic notation. Nevertheless, this informal semantics will become entirely precise when we give our algebraic semantics.

We should view a MOF metamodel specification definition as a pair $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, where $\widetilde{\mathcal{M}}$ is a metamodel definition, and $\widetilde{\mathcal{C}}$ is a set of OCL constraint definitions that any model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, must satisfy. What this specification describes is, of course, a *set* of models. We call this the *extensional* semantics of $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, and denote this semantics by $[\![(\mathcal{M}, \mathcal{C})]\!]_{\mathrm{MOF}}$. Recall that we use the notation $\widetilde{M} : \mathcal{M}$ for the conforms to relation, and $\widetilde{M} \models \mathcal{C}$ for the satisfaction of OCL constraints $\mathcal{C}$ by model $\widetilde{M}$. Using this notation, the extensional semantics can be informally defined as follows:

$$[\![(\mathcal{M}, \mathcal{C})]\!]_{\mathrm{MOF}} = \{\widetilde{M} \mid \widetilde{M} : \mathcal{M} \wedge \widetilde{M} \models \mathcal{C}\}.$$

5.2 A High-Level View of the MOF Metamodel Algebraic Semantics

We make the informal MOF semantics just described mathematically precise in terms of the *initial algebra semantics* of MEL. As already mentioned in Section 4, a MEL specification (Σ, E) has an associated initial algebra $T_{(\Sigma,E)}$. We call $T_{(\Sigma,E)}$ the *initial algebra semantics* of (Σ, E), and write

$$[\![(\Sigma, E)]\!]_{IAS} = T_{(\Sigma,E)}.$$

Let *SpecMOF* denote the set of all MOF metamodel specification definitions of the form $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, and let *SpecMEL* denote the set of all MEL specifications. The reason why we define *SpecMOF* as a set of pairs $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, instead than as a set of pairs $(\mathcal{M}, \mathcal{C})$ is because, as already mentioned, the mathematical status of $\mathcal{M}$ is, as yet, undefined, and is precisely one of the questions to be settled by a mathematical semantics. Instead, well-formed pairs $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ *are* data structures that can be syntactically characterized in a fully formal way. Therefore, the set *SpecMOF*, thus understood, is a well-defined mathematical entity. Our algebraic semantics is then defined as a mapping

$$\mathit{reflect} : \mathit{SpecMOF} \rightsquigarrow \mathit{SpecMEL}$$

that associates to each MOF metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ a corresponding MEL specification $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. The detailed definition of the mapping *reflect* will be given in following sections. But we can already make precise the way in which our informal semantics $[\![(\mathcal{M}, \mathcal{C})]\!]_{\mathrm{MOF}}$ is now made mathematically precise. Recall that any MEL signature Σ has an associated set S of sorts. Therefore, in the initial algebra $T_{(\Sigma,E)}$ each sort $s \in S$ has an associated set of elements $T_{(\Sigma,E),s}$. The key point is that in any MEL specification of the form $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, there is always a sort *ConsistentModelType*$\{\mathcal{M}\}$, which we also denote as $(\mathcal{M}, \mathcal{C})$ for short, whose data elements in the initial algebra are precisely the data representations of those models that both conform to $\mathcal{M}$ and satisfy $\mathcal{C}$. Therefore, we can give a precise mathematical semantics to our informal MOF extensional semantics by means of the equation

$$[\![(\mathcal{M}, \mathcal{C})]\!]_{\mathrm{MOF}} = T_{\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}), \mathit{ConsistentModelType}\{\mathcal{M}\}}.$$

Note that our algebraic semantics gives a precise mathematical meaning to the entities lacking such a precise meaning in the informal semantics, namely, the notions of: (i) metamodel $\mathcal{M}$, (ii) structural conformance relation $\widetilde{M} : \mathcal{M}$, (iii) metamodel realization $\mathit{reflect}(\widetilde{\mathcal{M}}, \emptyset)$, (iv) metamodel specification $(\mathcal{M}, \mathcal{C})$, (v) OCL constraint satisfaction relation $\widetilde{M} \models \mathcal{C}$, (vi) constrained conformance relation $\widetilde{M} : (\mathcal{M}, \mathcal{C})$, and (vi) metamodel specification realization $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. Specifically, a metamodel definition $\widetilde{\mathcal{M}}$ is defined as a precise mathematical object as the MEL theory $\mathit{reflect}(\widetilde{\mathcal{M}}, \emptyset)$, that is, the mathematical object defining $\widetilde{\mathcal{M}}$ is the MEL theory associated by *reflect* to $\widetilde{\mathcal{M}}$ when the set $\widetilde{\mathcal{C}}$ of OCL constraints is empty. In $\mathit{reflect}(\widetilde{\mathcal{M}}, \emptyset)$, there is always a sort *ModelType*$\{\mathcal{M}\}$, whose data elements in the initial algebra are precisely the data representations of those models that *only* conform to $\mathcal{M}$. The *structural conformance* relation between a model and its metamodel is then defined mathematically by the equivalence

$$\widetilde{M} : \mathcal{M} \quad \Leftrightarrow \quad \widetilde{M} \in T_{\mathit{reflect}(\widetilde{\mathcal{M}}, \emptyset), \mathit{ModelType}\{\mathcal{M}\}}.$$

Finally, the OCL constraint satisfaction relation acquires a precise mathematical meaning by means of the equivalence

$$\widetilde{M} \models \mathcal{C} \quad \Leftrightarrow \quad \widetilde{M} \in T_{reflect(\widetilde{\mathcal{M}},\widetilde{\mathcal{C}}),\, ConsistentModelType\{\mathcal{M}\}}.$$

We can, of course, combine these two relations into a more general conformance relation, i.e., the *constrained conformance* relation, in which a model conforms not just to a metamodel $\mathcal{M}$ but to a metamodel specification $(\mathcal{M}, \mathcal{C})$. This is defined by means of the equivalence

$$\widetilde{M} : (\mathcal{M}, \mathcal{C}) \quad \Leftrightarrow \quad \widetilde{M} : \mathcal{M} \ \wedge \ \widetilde{M} \models \mathcal{C}.$$

The MEL theory that is generated by the *reflect* mapping for a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ provides a mathematical meaning for both the metamodel definition $\widetilde{\mathcal{M}}$ and the set $\widetilde{\mathcal{C}}$ of OCL constraints that provide well-formedness requirements for such a metamodel. The OCL constraint satisfaction relation constitutes a powerful notion by means of which every model definition $\widetilde{M} : (\mathcal{M}, \mathcal{C})$ is inherently well-formed, conforming to the metamodel $\mathcal{M}$, $\widetilde{M} : \mathcal{M}$, and satisfying the constraints $\mathcal{C}$, $\widetilde{M} \models \mathcal{C}$. Note that, in the current OCL standard, OCL constraints can only be checked over specific model definitions $\widetilde{M}$, and there is no automated mechanism to formally categorize the set of well-formed models that conform to a metamodel that is enriched with OCL constraints, that is, to a metamodel specification $(\mathcal{M}, \mathcal{C})$. This mechanism is easily represented in the underlying MEL by means of the notion of membership, so that a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, only belongs to the carrier of the sort *ConsistentModelType*$\{\mathcal{M}\}$ iff $\widetilde{M} \models \mathcal{C}$, as we develop in next sections.

This formal characterization permits using models as first-class citizens, rising the level of abstraction of model-based tasks, where the internals of a specific model remain hidden. For example, a model transformation that is defined at level M2 between a source metamodel specification definition $(\widetilde{\mathcal{A}}, \widetilde{\mathcal{C}_{\mathcal{A}}})$ and a target metamodel specification definition $(\widetilde{\mathcal{B}}, \widetilde{\mathcal{C}_{\mathcal{B}}})$ can be mathematically defined as a function $f : [\![(\mathcal{A}, \mathcal{C}_{\mathcal{A}})]\!]_{\mathrm{MOF}} \rightarrow [\![(\mathcal{B}, \mathcal{C}_{\mathcal{B}})]\!]_{\mathrm{MOF}}$. Given a model definition $\widetilde{M} : (\mathcal{A}, \mathcal{C}_{\mathcal{A}})$, we can then use the model $f(\widetilde{M})$, where $f(\widetilde{M}) : (\mathcal{B}, \mathcal{C}_{\mathcal{B}})$ without any need for knowing the specific objects that constitute either $\widetilde{M}$ or $f(\widetilde{M})$. Note that, in addition, the sets $\mathcal{C}_{\mathcal{A}}$ and $\mathcal{C}_{\mathcal{B}}$ of OCL constraints are implicitly taken into account without any need for performing additional checking tasks.

The semantics of a metamodel $\mathcal{M}$ is provided by the set of all model definitions $\widetilde{M}$ that are well-defined regarding the types in $\mathcal{M}$, i.e., $\widetilde{M} : \mathcal{M}$. Therefore, $\mathcal{M}$ can also be interpreted as a sort whose values are terms that represent model definitions. At this point, we summarize the terminology that we use for the concepts that we develop in detail in the next sections:

- $\widetilde{\mathcal{M}}$: metamodel definition, or model type definition;
- $reflect(\widetilde{\mathcal{M}}, \emptyset)$: metamodel realization;
- $\mathcal{M}$: metamodel, or model type sort (in the MEL theory $reflect(\widetilde{\mathcal{M}}, \emptyset)$, $\mathcal{M}$ is represented by the *ModelType*$\{\mathcal{M}\}$ sort);
- $\widetilde{M} : \mathcal{M}$: structural conformance relation;
- $[\![\mathcal{M}]\!]_{\mathrm{MOF}}$: semantics of the metamodel $\mathcal{M}$;
- $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$: metamodel specification definition, or consistent model type definition;
- $reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$: metamodel specification realization;
- $(\mathcal{M}, \mathcal{C})$: metamodel specification, or consistent model type sort (in the MEL theory $reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, $(\mathcal{M}, \mathcal{C})$ is represented by the *ConsistentModelType*$\{\mathcal{M}\}$ sort);
- $\widetilde{M} \models \mathcal{C}$: OCL constraint satisfaction relation;
- $\widetilde{M} : (\mathcal{M}, \mathcal{C})$: constrained conformance relation; and
- $[\![(\mathcal{M}, \mathcal{C})]\!]_{\mathrm{MOF}}$: semantics of the metamodel specification $(\mathcal{M}, \mathcal{C})$.

5.3 Formalization of the MOF Reflection support

A complete formal support for reflection in the MOF framework involves: reflection and reification mappings, which permit working not only with a metamodel definition $\widetilde{\mathcal{M}}$ but also with its realization as a mathematical entity $\mathit{reflect}(\widetilde{\mathcal{M}}, \varnothing)$; and a mechanism to manipulate the metadata representation of metamodels and models in a generic way.

On the one hand, we provide an algebraic semantics for the reflection and reification mechanisms for MOF metamodels. The OCL standard mathematically defines a metamodel as an algebraic signature whose semantics is given in terms of domain theory [32]. In our approach, an alternative formalization is given by the computable, equationally-defined *reflect* function, which maps a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ into a theory in Membership Equational Logic. Our algebraic semantics associates a *different* MEL theory $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ to each different MOF metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. Note that in this algebraic semantics, the models conforming to such a MOF metamodel specification are precisely the elements $\widetilde{M} \in T_{\mathit{reflect}(\widetilde{\mathcal{M}},\widetilde{\mathcal{C}}),\mathit{ConsistentModelType}\{\mathcal{M}\}}$. Therefore, models conforming to different metamodel specifications belong to different algebraic data types. Far from performing a mere code generation task, this function provides the mathematical characterization of a metamodel, which is then interpreted as an initial algebra. This fact makes the formalization more practical, enabling the application of automated formal reasoning techniques for metamodels in the MOF framework.

On the other hand, the MOF standard provides a specification of the so-called MOF Reflection facilities. This focuses on the manipulation of the metarepresentation of a MOF metamodel definition $\widetilde{\mathcal{M}}$. In particular, in the MOF metamodel the `Object` object type provides an API to query and manipulate MOF objects. Any other MOF object type, including the `Class` object type, specializes the `Object` object type and therefore inherits this API. This feature is very useful for MOF metamodels. The reification $\widetilde{\mathcal{M}}$ of a MOF metamodel realization $\mathit{reflect}(\widetilde{\mathcal{M}}, \varnothing)$ is a collection of MOF objects and therefore, by inheritance, a collection of objects of class `Object`. For purposes of the MOF Reflection facilities, we are interested in a *single* data representation for models, which can be manipulated with a single API, the MOF Object object type. The way this is formalized in our algebraic semantics is as follows. For each MOF metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the MEL theory $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ contains a shared subtheory, called `META-MODEL`, whose data elements are not modified by the theory inclusion `META-MODEL` $\subseteq \mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. The single, metamodel-independent representation of a model $\widetilde{M}$ is provided by the elements of a sort $\mathit{ModelType}\{\mathit{MetaObject}\}$ in `META-MODEL`. We use $\widehat{M}$ to denote this metamodel-independent representation of $\widetilde{M}$. Specifically, terms $\widehat{M}$ are sets of terms of sort *MetaObject*, which metarepresent the explicitly typed objects that make up the model definition $\widetilde{M}$. The change of representation $\widetilde{M} \mapsto \widehat{M}$ and its inverse $\widehat{M} \mapsto \widetilde{M}$ are supported in the MEL theory $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ by two equationally-defined operations

$$\mathit{upModel} : \mathit{ConsistentModelType}\{\mathcal{M}\} \longrightarrow \mathit{ModelType}\{\mathit{MetaObject}\}$$

$$\mathit{downModel} : \mathit{ModelType}\{\mathit{MetaObject}\} \rightsquigarrow \mathit{ConsistentModelType}\{\mathcal{M}\}.$$

where the second operation is partial, but becomes total at the level of the corresponding kinds (see Section 4). For each model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, these two operations satisfy the equations $\mathit{downModel}(\mathit{upModel}(\widetilde{M})) = \widetilde{M}$ and $\mathit{upModel}(\mathit{downModel}(\widehat{M})) = \widehat{M}$.

The `META-MODEL` theory, together with its subtheory inclusion in each theory $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, formally defines the notion of MOF Reflection Facilities, that permit the manipulation of the metarepresentation of model definitions $\widetilde{M}$ and metamodel definitions $\widetilde{\mathcal{M}}$. This theory introduces an extra sublevel in the metalevels M1, M2, and M3 of the MOF Framework. A metamodel realization $\mathit{reflect}(\widetilde{\mathcal{M}}, \varnothing)$ can be metarepresented as $\widetilde{\mathcal{M}}$, which can be manipulated by means of specific operators of the MOF metamodel. Taking the `META-MODEL` theory into account, the objects that constitute $\widetilde{\mathcal{M}}$ can be represented as instances of the MOF Object object type in $\widehat{\mathcal{M}}$ by means of the *upModel* operator. $\widehat{\mathcal{M}}$ can be manipulated by means of the methods of the MOF Object object type by means of a function $\widehat{\phi}$. This process is summarized in Fig. 5.1. The MOF Object level introduces a type-agnostic layer, where objects can be manipulated by means of a generic API. Note

$$\begin{tikzcd}
\textit{metadata level} & \textit{untyped sublevel} & \widehat{\mathcal{M}} \arrow[r, mapsto, "\widehat{\phi}"] & \widehat{\mathcal{M}'} \arrow[d, mapsto, "\mathit{downModel}"] \\
 & \textit{typed sublevel} & \widetilde{\mathcal{M}} \arrow[u, mapsto, "\mathit{upModel}"] \arrow[r, mapsto, "\widetilde{\phi}"] & \widetilde{\mathcal{M}'} \arrow[d, mapsto, "\mathit{reflect}"] \\
\textit{base level} & & \mathit{reflect}(\widetilde{\mathcal{M}}, \varnothing) \arrow[u, mapsto, "\mathit{reify}"] \arrow[r, mapsto, "\phi"] & \mathit{reflect}(\widetilde{\mathcal{M}'}, \varnothing)
\end{tikzcd}$$

Figure 5.1: M2 sublevels.

$$\begin{tikzcd}
\text{MOF } \textit{Object level} & \widehat{M} \arrow[r, mapsto, "\widehat{\phi}"] & \widehat{M'} \arrow[d, mapsto, "\mathit{downModel}"] \\
\textit{metadata level} & \widetilde{M} \arrow[u, mapsto, "\mathit{upModel}"] \arrow[r, mapsto, "\widetilde{\phi}"] & \widetilde{M'}
\end{tikzcd}$$

Figure 5.2: M1 sublevels.

that both metadata sublevels, untyped and typed, in a specific metalevel of the MOF framework appear specified in the MOF standard in an informal way. Our aim here is to provide a simple but precise and powerful algebraic definition of such constructs.

In the MOF reflection process supported by *reflect*, each object in the collection $\widetilde{\mathcal{M}}$ that is an instance of the MOF `Class` object type is mapped to an object type in $\mathit{reflect}(\widetilde{\mathcal{M}}, \varnothing)$ that specializes the algebraic representation of the MOF `Object` object type, thus inheriting its API. This means that any object in a model definition $\widetilde{M}$ that conforms to a MOF metamodel $\mathcal{M}$ can be queried and transformed by means of this reflective API, as shown in Fig. 5.2. For example, we can transform a UML model by adding or deleting properties to some classes in that model. The algebraic semantics of the MOF Reflection facilities is developed in the `META-MODEL` theory, which is presented in more detail in Section 8.

5.4 Discussion about the Algebraic Semantics of MOF Metamodels

In this section, we introduce the structure that is used in Sections 6-8 to describe an algebraic semantics of the MOF framework. Throughout these sections we present an automated mechanism to define the algebraic semantics of a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ by means of the *reflect* function. The *reflect* function provides a set of formal notions for $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$: (i) metamodel $\mathcal{M}$, (ii) structural conformance relation $\widetilde{M} : \mathcal{M}$, (iii) metamodel realization $\mathit{reflect}(\widetilde{\mathcal{M}}, \varnothing)$, (iv) metamodel specification $(\mathcal{M}, \mathcal{C})$, (v) OCL constraint satisfaction relation $\widetilde{M} \models \mathcal{C}$, (vi) constrained conformance relation $\widetilde{M} : (\mathcal{M}, \mathcal{C})$, and (vi) metamodel specification realization $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. In addition, we provide an algebraic semantics for the MOF Reflection Facilities, which together with the *reflect* and *reify* functions constitutes a complete formal support for reflection in a MOF framework, which is informally supported in current MOF implementations.

These concepts are developed in Sections 6-8. However, their definitions are interrelated, leading to many cross-references between these Sections. In addition, some of these concepts are defined in a self-referential way, due to the reflective character of the MOF Framework. In this section, we identify these self-referential definitions, providing the building blocks that permit bootstrapping a formal MOF framework in several stages:

1. The function
$$\mathit{reflect} : \mathit{SpecMOF} \rightsquigarrow \mathit{SpecMEL}$$
provides the algebraic semantics for the types that are defined in a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. We define the algebraic data type $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$, whose values are well-formed MOF metamodel definitions $\widetilde{\mathcal{M}}$, without taking into account the OCL constraints of the MOF metamodel. To define the *reflect* function, we first define a partial function
$$\mathit{reflect}_{\mathrm{MOF}} : [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \rightsquigarrow \mathit{SpecMEL},$$
satisfying the equation $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}) = \mathit{reflect}(\widetilde{\mathcal{M}}, \varnothing)$. On the one hand, the metamodel definition $\widetilde{\mathrm{MOF}}$ is a special case of metamodel definition, i.e., $\widetilde{\mathrm{MOF}} \in [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$. On the other hand, the $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ domain is defined as data in the metamodel definition $\widetilde{\mathrm{MOF}}$, and is formally defined by the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory. Therefore, we identify a self-referential definition of the $\mathit{reflect}_{\mathrm{MOF}}$ function: **the domain of the $\mathit{reflect}_{\mathbf{MOF}}$ function is defined by the function itself!** We break this self-referential definition by first defining the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory, which constitutes the first building block for our MOF framework. Once the data type $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ is defined in this way, we then define the $\mathit{reflect}_{\mathrm{MOF}}$ function for any metamodel definition $\widetilde{\mathcal{M}} \in [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$. The $\mathit{reflect}_{\mathrm{MOF}}$ function is discussed in more detail in Section 6.

2. Without taking the semantics of OCL constraints into account, a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ is constituted by a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}} \in [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$, and by a set $\widetilde{\mathcal{C}}$ of OCL constraints $\widetilde{c}$, such that $\widetilde{c} \in [\![\mathrm{OCL}]\!]_{\mathrm{MOF}}$, where $[\![\mathrm{OCL}]\!]_{\mathrm{MOF}}$ is an algebraic data type whose elements are well-formed OCL expressions. This data type is defined as data using the metamodel definition $\widetilde{\mathrm{OCL}}$, given that $\widetilde{\mathrm{OCL}} \in [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$. Taking into account the semantics of OCL constraints, we define two algebraic data types: $[\![(\mathrm{MOF}, \mathcal{C}_{\mathrm{MOF}})]\!]_{\mathrm{MOF}}$ as the set of metamodel definitions $\widetilde{\mathcal{M}}$ that satisfy the following conditions: $\widetilde{\mathcal{M}} \in [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ and $\widetilde{\mathcal{M}} \models \mathcal{C}_{\mathrm{MOF}}$; and $[\![(\mathrm{OCL}, \mathcal{C}_{\mathrm{OCL}})]\!]_{\mathrm{MOF}}$ as the set of OCL expressions $\widetilde{c}$ that satisfy the following conditions: $\widetilde{c} \in [\![\mathrm{OCL}]\!]_{\mathrm{MOF}}$ and $\widetilde{c} \models \mathcal{C}_{\mathrm{OCL}}$. The *reflect* function can then be defined as a function:
$$\mathit{reflect} : [\![(\mathrm{MOF}, \mathcal{C}_{\mathrm{MOF}})]\!]_{\mathrm{MOF}} \times \mathcal{P}_{\mathit{fin}}([\![(\mathrm{OCL}, \mathcal{C}_{\mathrm{OCL}})]\!]_{\mathrm{MOF}}) \rightsquigarrow \mathit{SpecMEL}.$$
The *reflect* function uses the semantics for the OCL constraint satisfaction relations $\widetilde{\mathcal{M}} \models \mathcal{C}_{\mathrm{MOF}}$, in the $\mathit{reflect}(\widetilde{\mathrm{MOF}}, \widetilde{\mathcal{C}_{\mathrm{MOF}}})$ theory, and the semantics of the satisfaction relation $\widetilde{c} \models \mathcal{C}_{\mathrm{OCL}}$ between an OCL constraint and the constraints $\widetilde{\mathcal{C}_{\mathrm{OCL}}}$ associated to the OCL metamodel in the $\mathit{reflect}(\widetilde{\mathrm{OCL}}, \widetilde{\mathcal{C}_{\mathrm{OCL}}})$ theory. However, in the latter case, OCL constraints $\widetilde{c}$ in $\widetilde{\mathcal{C}_{\mathrm{OCL}}}$ are again model definitions $\widetilde{c}$ such that $\widetilde{c} \in [\![(\mathrm{OCL}, \mathcal{C}_{\mathrm{OCL}})]\!]_{\mathrm{MOF}}$. At this point, we find the second self-referential definition: **the domain of the *reflect* function relies on a set $\widetilde{\mathcal{C}_{\mathbf{OCL}}}$ of OCL constraints whose semantics is provided by the *reflect* function itself!** Recall the sort $\mathit{ConsistentModelType}\{\mathcal{M}\}$ in the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory, so that
$$[\![(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})]\!]_{\mathrm{MOF}} = T_{\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}), \mathit{ConsistentModelType}\{\mathcal{M}\}}.$$
In Section 7, we indicate how the semantics of OCL expressions is defined in the *reflect* function. This function permits constraining the domain $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ into the smaller domain $[\![(\mathrm{MOF}, \mathcal{C}_{\mathrm{MOF}})]\!]_{\mathrm{MOF}}$ so that
$$[\![(\mathrm{MOF}, \mathcal{C}_{\mathrm{MOF}})]\!]_{\mathrm{MOF}} \subseteq [\![\mathrm{MOF}]\!]_{\mathrm{MOF}},$$
and the domain $[\![\mathrm{OCL}]\!]_{\mathrm{MOF}}$ into the smaller domain $[\![(\mathrm{OCL}, \mathcal{C}_{\mathrm{OCL}})]\!]_{\mathrm{MOF}}$ so that
$$[\![(\mathrm{OCL}, \mathcal{C}_{\mathrm{OCL}})]\!]_{\mathrm{MOF}} \subseteq [\![\mathrm{OCL}]\!]_{\mathrm{MOF}}.$$

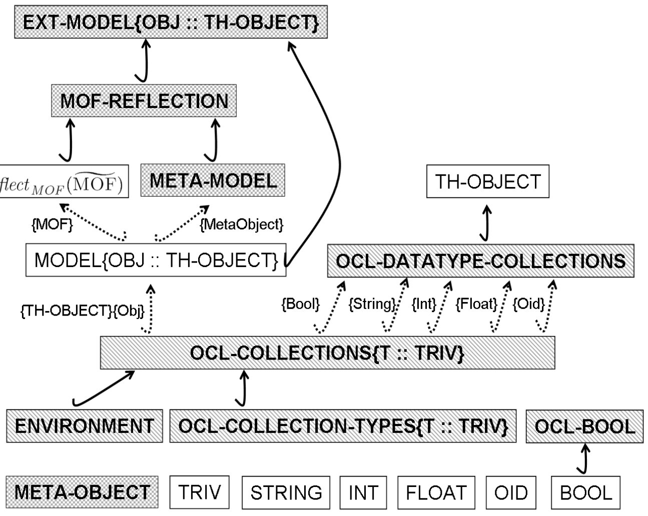


Figure 5.3: Infrastructure of parameterized theories.

3. Each model definition $\widetilde{M}$, such that $\widetilde{M} : (\mathcal{M}, \mathcal{C})$, which is defined at the base level of a specific conceptual level in the MOF framework, can be queried and manipulated by means of a generic API, called *MOF Reflection Facilities*, which is provided in a theory called `META-MODEL`. To manipulate the model definition $\widetilde{M}$ by means of this API, we have to metarepresent it at the metalevel of the corresponding conceptual level of the MOF framework by means of the *upModel* function, presented in the previous section. There is a sort *ModelType*{*MetaObject*} in the `META-MODEL` theory whose values are the metarepresentation $\widehat{M}$ of any model definition $\widetilde{M}$.

   The *reflect*$_{\mathrm{MOF}}$ function could be generically defined by using the domain $T_{\texttt{META-MODEL},ModelType\{MetaObject\}}$ instead of $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$. However, we prefer using the *reflect*$_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory for bootstrapping the mechanical formalization of the MOF framework. Using the `META-MODEL` theory as the bootstrapping theory to define the *reflect*$_{\mathrm{MOF}}$ function for a metamodel definition $\widetilde{\mathcal{M}}$ implies would require with the metarepresentation of the types in $\mathcal{M}$ at a base level, which have not been defined yet. Therefore, we first present the algebraic semantics of the metamodels MOF and OCL at a base level in Section 6 and 7 by means of the *reflect* function. After explaining how the semantics of both metamodels can be metarepresented by means of the *upModel* function in Section 8, we indicate how the *reflect* function can be specified at the object metalevel, simplifying the formalization approach from a more generic point of view.

To break the self-reference in the definition of the *reflect* function, we usually follow a two-step strategy where: (a) the *reflect*$_{\mathrm{MOF}}$ function is first defined for the specific metamodel definition $\widetilde{\mathrm{MOF}}$, which permits defining the $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ domain; and (b) the second step consists in the generalization of the *reflect*$_{\mathrm{MOF}}$ function for any metamodel definition $\widetilde{\mathcal{M}}$. The *reflect*$_{\mathrm{MOF}}$ function is then used to define the *reflect* function. A *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory has a generic part, independent from any metamodel definition $\widetilde{\mathcal{M}}$, and a metamodel-specific part, whose definition depends on $\widetilde{\mathcal{M}}$ and $\widetilde{\mathcal{C}}$. To avoid defining the generic part of the theory twice $-$ a first time for the metamodel definition $\widetilde{\mathrm{MOF}}$, and a second time for any metamodel definition $\widetilde{\mathcal{M}}$ $-$, we have specified it by means of a set of parameterized theories that will be instantiated depending on the metamodel definition $\widetilde{\mathcal{M}}$. These theories are shown in Fig. 5.3, and are further explained throughout Sections 6-8. They can be summarized as follows:

1. *Types.* The theories that provide the sorts and constructors to define the algebraic semantics of the types that are defined in a metamodel definition $\widetilde{\mathcal{M}}$ are depicted in white. The theories

`STRING`, `BOOL`, `INT`, `FLOAT`, and `OID` provide the predefined basic data types, and the parameterized theory `OCL-COLLECTION-TYPES{X :: TRIV}` provides the parameterized OCL collection types. Although the `BOOL` theory is predefined, some of its operators are redefined to match the semantics of the operators for the OCL Boolean primitive type, in Section 7. The `MODEL{OBJ :: TH-OBJECT}` theory provides the constructors that are needed to define objects and model definitions as collections of objects. The `EXT-MODEL{OBJ :: TH-OBJECT}` theory includes the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory, without changing any of its type definitions, so that the $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ type can be used in it. The theories that permit defining the type semantics for a metamodel definition $\widetilde{\mathcal{M}}$ are presented in more detail in Section 6.

2. *Operators.* Once the $\mathit{reflect}_{\mathrm{MOF}}$ function is defined, the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{OCL}})$ theory provides the semantics of the types that are defined in the metamodel definition $\widetilde{\mathrm{OCL}}$. In a second stage, we define the operators for all the types that can be used in OCL expressions in order to define OCL constraints. The theories that are depicted with a dashed background provide these predefined operators, and are discussed in more detail in Section 7.

3. *MOF Reflection Facilities.* In a third stage, we present the theories that provide the MOF Reflection Facilities API to manipulate model definitions $\widetilde{M}$, such that $\widetilde{M} : (\mathcal{M}, \mathcal{C})$, by means of a generic API. These theories, which are depicted with dotted background in Fig. 5.3, constitute together with the *reflect* and *reify* functions a complete support for reflection in a MOF framework. These theories are presented in Section 8.

Chapter 6

An Algebraic Structural Conformance Relation

As introduced in the previous section, the *reflect* function maps a MOF metamodel specification $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ to a MEL theory $(S, <, \Omega, E \cup A)$. This theory provides a *hierarchy of algebraic types*, where the hierarchy itself is defined by a partially ordered set of sorts $(S, \leqslant)$, where $\leqslant$ is defined as the reflexive closure of the strict partial order $<$ of subsort inclusion relationships, and by a set of operator declarations Ω, whose semantics is defined by the set of conditional equations and memberships $E \cup A$. The set Ω contains constructor operators to define values of sorts in S, and defined functions over the sorts in S, i.e., operators that disappear during the equational simplification process modulo the equations and memberships in $E \cup A$. In a type system, *type safety* is the guarantee that no runtime-error will result from the application of a defined function to the wrong value. A *type system* is a set of rules for checking type safety, i.e., the *isValueOf* relation between a value and its type. Under appropriate requirements on Ω and $E \cup A$, which are met for *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, each value that can appear in a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, always has at *least one* sort, and always a *smallest possible* sort in the hierarchy of sorts provided by $(S, \leqslant)$.

In our approach, the types that are provided by *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ are not only syntactically defined in a MEL theory *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, but also semantically defined using the initial algebra that is directly associated to it[1]. In the initial algebra, the formal semantics of a specific type is given by the carrier of the corresponding sort in S (or value domain), and the formal semantics of the *isValueOf* relation is given by the membership relation of a specific value to the carrier of the corresponding sort. Among the types that are provided by the $reflect(\widetilde{M}, \widetilde{\mathcal{C}})$ theory, we find *object types* and, what is more relevant, *model types*. While the *isValueOf* relation is refined as the *instanceOf* relation for an object type, it is instead refined as the *structural conformance* relation for a model type. A model type permits a syntactic characterization of a model definition $\widetilde{M}$ itself as a collection of objects, and a semantic characterization of the set of *conformant* model definitions $\widetilde{M}$ such that $\widetilde{M} : (\mathcal{M}, \mathcal{C})$. Therefore, model definitions $\widetilde{M}$ can be treated as first-class citizens in both a practical and a formal way.

To define the *reflect* function, we consider the *SpecMOF* data type, whose elements are metamodel specification definitions $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. In the end, the function we want to define is

$$reflect : SpecMOF \rightsquigarrow SpecMEL.$$

This function provides the algebraic semantics for the types that are defined in a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. Both $\widetilde{\mathcal{M}}$ and each constraint definition $\widetilde{c}$ in $\widetilde{\mathcal{C}}$ can be defined as models taking into account the algebraic semantics that is provided for both metamodel definitions $\widetilde{\mathrm{MOF}}$ and $\widetilde{\mathrm{OCL}}$, respectively. The algebraic semantics of both metamodels is provided by the *reflect* func-

[1]This initial algebra has a very simple description in terms of canonical forms for the equations E modulo A, given that the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory is ground confluent, terminating and pre-regular.

tion, again! To break the self-reference in the definition of the *reflect* function, we first define a simpler function

$$\mathit{reflect}_{\mathrm{MOF}} : [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \rightsquigarrow \mathit{SpecMEL},$$

satisfying the equation $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}) = \mathit{reflect}(\widetilde{\mathcal{M}}, \varnothing)$. $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ is the data type whose elements are metamodel definitions $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}} : \mathrm{MOF}$. For elements $\widetilde{\mathcal{M}}$ in $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$, the satisfaction of OCL constraints $\widetilde{\mathcal{C}_{\mathrm{MOF}}}$ is not mandatory. The definition of the $\mathit{reflect}_{\mathrm{MOF}}$ function still has the same self-reference problem since $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ is defined by means of the $\mathit{reflect}_{\mathrm{MOF}}$ function itself. In subsequent sections, we define the $\mathit{reflect}_{\mathrm{MOF}}$ function as follows:

1. The MEL theory that provides the algebraic semantics for the types of a specific metamodel can be split into a generic part, common to all metamodels, and a part specific to the given metamodel. To avoid presenting the generic part of a metamodel specification twice: (a) a first time for the metamodel definition $\widetilde{\mathrm{MOF}}$, and (b) a second time for any metamodel definition $\widetilde{\mathcal{M}}$; we present it as a set of parameterized MEL theories. These theories provide the semantics of the primitive data types and the OCL collection types that can be used in the MOF Framework.

2. Recall that $\widetilde{\mathrm{MOF}}$ is itself a MOF metamodel, since $\widetilde{\mathrm{MOF}} : \mathrm{MOF}$, as explained in Section 3.2. We first define a MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$, that is, we first define $\mathit{reflect}_{\mathrm{MOF}}$ for a *single* metamodel, namely $\widetilde{\mathrm{MOF}}$. In particular, $\mathit{reflect}_{\mathrm{MOF}}$ instantiates the aforementioned parameterized theories for the metamodel definition $\widetilde{\mathrm{MOF}}$. The $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory defines the $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ type as the set of metamodel definitions $\widetilde{\mathcal{M}}$, which can be viewed as both graphs and trees. This step breaks the self-reference in the $\mathit{reflect}_{\mathrm{MOF}}$ function definition, constituting the first solid building block on which to define the *reflect* function.

3. Once the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory, such that

   $$\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}) = \mathit{reflect}(\widetilde{\mathrm{MOF}}, \varnothing),$$

   is defined, we identify the $\mathit{ModelType}\{\mathrm{MOF}\}$ sort in this theory, whose carrier in the initial algebra defines the $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ type, i.e.,

   $$[\![\mathrm{MOF}]\!]_{\mathrm{MOF}} = T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{ModelType}\{\mathrm{MOF}\}}.$$

   Note that, in particular, this means that

   $$\widetilde{\mathrm{MOF}} \in [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}.$$

   We then define the value of the function $\mathit{reflect}_{\mathrm{MOF}}$ on *any* metamodel $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}} \in [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$, as its corresponding MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$. In this case, the parameterized theories of step 1 are instantiated for the metamodel definition $\widetilde{\mathcal{M}}$. Given a metamodel definition $\widetilde{\mathcal{M}}$, the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory defines the $[\![\mathcal{M}]\!]_{\mathrm{MOF}}$ type as the set of model definitions $\widetilde{M}$ that are constituted by a collection of typed objects, which can be viewed as both a graph and a tree. However, note that the $[\![\mathcal{M}]\!]_{\mathrm{MOF}}$ type does not consider the OCL constraint satisfaction relation yet. The constrained conformance relation is studied in Section 7.

4. At the end of this section, a brief description of how the $\mathit{reflect}_{\mathrm{MOF}}$ function has been embedded into the MEL reflective semantics is provided, and a brief description of a *reify* function that permits obtaining the metamodel definition $\widetilde{\mathcal{M}}$ from the MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$, such that $\widetilde{\mathcal{M}} = \mathit{reify}(\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}))$, is also provided.

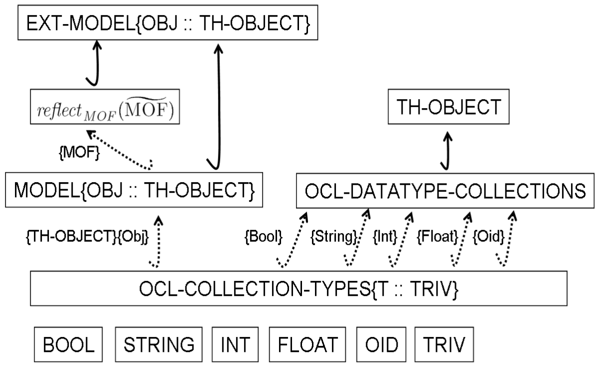


Figure 6.1: Infrastructure of parameterized theories.

6.1 A Generic Infrastructure of Parameterized Theories

In this section, we describe the set of theories, some of them parameterized, that provide the algebraic semantics for the predefined types in a MOF metamodel: primitive types and OCL collection types. Fig. 6.1 shows a simplification of the complete structure of theories that is shown in Fig. 5.3, namely, those theories used to define the algebraic semantics of the types in a metamodel realization $reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$. In this section, we only present the constructors that are needed to define values for these types. Type operators are presented in Section 7 in order to define OCL expressions.

In subsequent sections, we detail the theories that appear in Fig. 6.1, which are used to define the $reflect_{\mathrm{MOF}}$ function:

- *Basic data type theories:* Provide the semantics for the OCL primitive types.
- *Parameter theories:* Provide the formal parameters for the parameterized theories `OCL-COLLECTION-TYPES`, `MODEL` and `EXT-MODEL`.
- `OCL-COLLECTION-TYPES` *theory:* Provides the semantics of OCL collection types, which can be instantiated for primitive data types, enumeration types and object types.
- `MODEL` *theory:* Provides the constructors that are needed to define objects and model definitions. This theory is only instantiated for the metamodel definition $\widetilde{\mathrm{MOF}}$ in the $reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory. The $reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory is presented in Section 6.2, defining the model type $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$.
- `EXT-MODEL` *theory:* Provides the constructors to define objects and model definitions for any MOF metamodel definition $\widetilde{\mathcal{M}}$. This theory is instantiated for a specific metamodel definition $\widetilde{\mathcal{M}}$, different from $\widetilde{\mathrm{MOF}}$, in a theory $reflect_{MOF}(\widetilde{\mathcal{M}})$, which provides the model type $[\![\mathcal{M}]\!]_{\mathrm{MOF}}$. The semantics of a metamodel $\mathcal{M}$ is provided in Section 6.3.

6.1.1 Primitive Type Theories

Basic data types are implemented in Maude as built-in types, and their algebraic semantics is defined in [75] (see Chapter 9). More specifically, the `BOOL` theory provides the sort `Bool`, the `STRING` theory provides the sort `String`, the `INT` theory provides the sort `Int`, and the `FLOAT` theory provides the sort `Float`. In addition, we define the `OID` theory as

```
fmod OID is
    sorts Oid .
    op nullOid : -> [Oid] .
endfm
```

This theory provides the sort `Oid` that represents the object identifier type.

These sorts may be used to define the type of object properties. Properties of this kind may not require a value so that a *null* value can be used. This *null* value is not explicitly represented in any of the theories `BOOL`, `STRING`, `INT`, nor `FLOAT`. In our approach, a *null* value for a specific sort T is identified with a term in the kind $[T]$. A null value belongs to a kind [T] but not to a specific sort T. Null values can be defined by means of constants that are defined, in Maude notation, as follows:

```
op nullBool : -> [Bool] .
op nullString : -> [String] .
op nullInt : -> [Int] .
op nullFloat : -> [Float] .
```

These constants are defined in the `OCL-DATATYPE-COLLECTIONS` theory.

In MOF there are four basic data types that are reused from the UML specification: Boolean, String, Integer and UnlimitedNatural. Instead of these types, we use the basic data types of the OCL specification to align the MOF algebraic semantics with the OCL algebraic semantics, that is, we consider the basic types: Boolean, String, Integer and Real. The algebraic semantics of the four primitive types that can be used in the MOF framework is provided as follows:

$$\boxed{\begin{array}{l} [\![\textsc{Boolean}]\!]_{\mathrm{MOF}} = T_{\texttt{OCL-DATATYPE-COLLECTIONS},Bool} \\ [\![\textsc{String}]\!]_{\mathrm{MOF}} = T_{\texttt{OCL-DATATYPE-COLLECTIONS},String} \\ [\![\textsc{Integer}]\!]_{\mathrm{MOF}} = T_{\texttt{OCL-DATATYPE-COLLECTIONS},Int} \\ [\![\textsc{Real}]\!]_{\mathrm{MOF}} = T_{\texttt{OCL-DATATYPE-COLLECTIONS},Float} \end{array}}$$

and the *isValueOf* relation between a value $\widetilde{v}$ and its corresponding primitive type PT is then formally defined by the equivalence

$$\boxed{\widetilde{v} : \mathrm{PT} \quad \Leftrightarrow \quad \widetilde{v} \in [\![\mathrm{PT}]\!]_{\mathrm{MOF}}}$$

Note that if $\widetilde{v}$ is a null value, it is not considered as a value of the corresponding type. The reason for this is that a property that is set to a null value is considered to be undefined (See section A.1.1.1 in [24]). We discuss the algebraic semantics of undefined values in Section 7.

6.1.2 OCL Collection Types

The theory `OCL-COLLECTION-TYPES` provides the sorts and constructors that permit defining OCL collections. The formal parameter of this theory is the trivial theory `TRIV` defined, in Maude notation, as follows:

```
fth TRIV is
  sorts Elt .
endfth
```

The `TRIV` theory only contains one sort, namely, `Elt`. OCL collection types are then defined in the MEL theory `OCL-COLLECTION-TYPES{T :: TRIV}`, which provides OCL parameterized sorts and constructors w.r.t. a generic parameter `T`. This theory is instantiated with a view, usually also called `T`, that is defined between the `TRIV` theory and the theory that provides the sort `T`. For example, to define collections of integers we need to instantiate it with the view

```
view Int from TRIV to INT as
  sort Elt to Int .
endv
```

by means of the expression `OCL-COLLECTION-TYPES{Int}`. In the same way, we define views for the theories `BOOL`, `STRING`, `FLOAT` and `OID`.

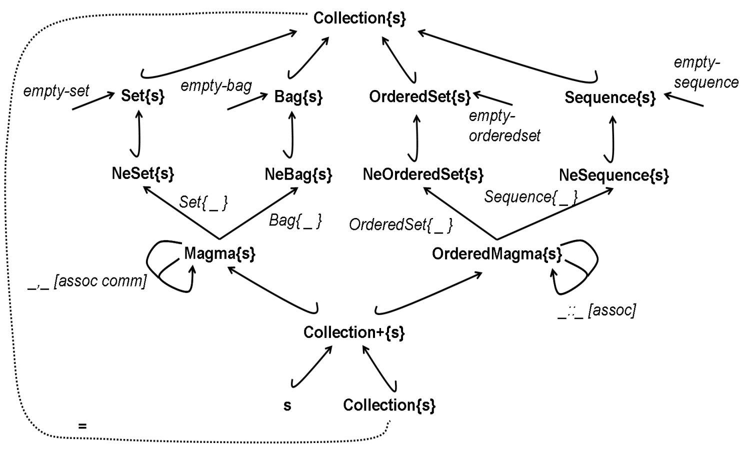


Figure 6.2: Subsort Structure of our Specification of the OCL Type System.

The `OCL-COLLECTIONS{T}` theory provides a set of generic sorts that represent OCL collections of elements that cannot be empty: `NeSet{T}`, `NeOrderedSet{T}`, `NeBag{T}` and `NeSequence{T}`. `NeSet{T}` represents a parameterized collection sort that is instantiated with the sort `T`. For example, we can define the sort `NeSet{Int}` for non-empty sets of integers.

To take into account the uniqueness and order features of an OCL collection, we introduce two intermediate sorts and their constructors (shown in Fig. 6.2): `Magma{T}` and `OrderedMagma{T}`. Basically, we define the sort `Magma{T}` as the sort whose terms represent multisets of elements that are not ordered. The constructor for this sort has the symbol "_,_" and is defined as associative and commutative. Through the subsort relationships

`T < Collection+{T} < Magma{T}`,

constructors that can be used to define terms of sort `T` can also be used to define `Magma{T}` terms. Thus, working with integers, `1 , 2 , 3` is a term that represents a well-formed `Magma{Int}`. In addition, we can state that `1 , 2 , 3` and `3 , 2 , 1` represent the same set of elements modulo the commutative and associative attributes. Terms of the `OrderedMagma{T}` sort represent ordered lists of `s` terms that are built by means of the associative constructor "_::_". In this case, both `1 :: 2 :: 3` and `3 :: 2 :: 1` are well-formed `OrderedMagma{Int}` terms, but they are not equal, since "_::_" is associative but not commutative.

The four collection types, *Set*, *OrderedSet*, *Bag* and *Sequence*, are represented as parameterized sorts. For example, `Set{Int}` is the sort for sets of integers. In addition, each collection has a constructor operator, whose symbol coincides with that of the corresponding collection sort. For example,

`Set{_} : Magma{Int} → Set{Int}`

is the constructor symbol for Set collections of integers and it can be used to define, for example, the set `Set{1,2}`. `Magma{T}` terms are used to define sets and bags, while `OrderedMagma{T}` terms are used to define ordered sets and sequences. All these four types are subtypes of the `Collection{T}` type, which does not have a direct constructor operator. `Collection{T}` terms can only be created by means of constructors in its subsorts. In OCL, nested collections are allowed, so that an element in a set of integers can be another collection of integers. This is specified by indicating that one collection can be an element of another collection, i.e., by the subsort inclusion

`Collection{T} < Collection+{T}`.

For example, `Set{Sequence{1,2}, Sequence{2}}` is a valid term of type `Set{Sequence{Integer}}`.

The parameterized collection types, i.e., `Set{T}`, `OrderedSet{T}`, `Bag{T}`, and `Sequence{T}`, represent collections that can be empty for properties that have lower multiplicity = 0. Their carriers are those of the analogous collection types (`NeSet{T}`, `NeOrderedSet{T}`, `NeBag{T}` and `NeSequence{T}`) plus a value that represents an empty collection: `empty-set`, `empty-orderedset`, `empty-bag` and `empty-sequence`, respectively. The key sorts, subsorts, and constructor operators of the resulting equational theory that represents the OCL collections types is summarized, in Maude notation, as follows:

```
sorts NeSet{T} Set{T} NeOrderedSet{T} OrderedSet{T}
    NeBag{T} Bag{T} NeSequence{T} Sequence{T}
    Collection{T} Collection+{T} .
sorts Magma{T} OrderedMagma{T} .

subsort NeSet{T} < Set{T} .
subsort NeOrderedSet{T} < OrderedSet{T} .
subsort NeBag{T} < Bag{T} .
subsort NeSequence{T} < Sequence{T} .
subsorts Set{T} OrderedSet{T} Bag{T} Sequence{T} < Collection{T} .
subsorts T Collection{T} < Collection+{T} .
subsorts Collection+{T} < Magma{T} OrderedMagma{T} .

op _,_ : Magma{T} Magma{T} -> Magma{T} [ctor assoc comm] .
op _::_ : OrderedMagma{T} OrderedMagma{T} -> OrderedMagma{T} [ctor assoc] .

op empty-set : -> Set{T} .
op Set{_} : Magma{T} -> NeSet{T} [ctor] .

op empty-orderedset : -> OrderedSet{T} .
op OrderedSet{_} : OrderedMagma{T} -> NeOrderedSet{T} [ctor] .

op empty-bag : -> Bag{T} .
op Bag{_} : Magma{T} -> NeBag{T} [ctor] .

op empty-sequence : -> Sequence{T} .
op Sequence{_} : OrderedMagma{T} -> NeSequence{T} [ctor] .
```

The algebraic semantics of parameterized OCL collection types is given, in the `OCL-COLLECTIONS{T :: TRIV}` theory, by the following equations:

$$\boxed{\begin{array}{ll}
[\![\mathit{Set}\{T\}]\!]_{\mathrm{MOF}} & = T_{\texttt{OCL-COLLECTIONS\{T\}},\mathit{Set}\{T\}} \\
[\![\mathit{OrderedSet}\{T\}]\!]_{\mathrm{MOF}} & = T_{\texttt{OCL-COLLECTIONS\{T\}},\mathit{OrderedSet}\{T\}} \\
[\![\mathit{Bag}\{T\}]\!]_{\mathrm{MOF}} & = T_{\texttt{OCL-COLLECTIONS\{T\}},\mathit{Bag}\{T\}} \\
[\![\mathit{Sequence}\{T\}]\!]_{\mathrm{MOF}} & = T_{\texttt{OCL-COLLECTIONS\{T\}},\mathit{Sequence}\{T\}} \\
[\![\mathit{Collection}\{T\}]\!]_{\mathrm{MOF}} & = T_{\texttt{OCL-COLLECTIONS\{T\}},\mathit{Collection}\{T\}}
\end{array}}$$

where T is the formal parameter of the parameterized OCL collection types, which can be represented as a view that maps the `TRIV` theory to the corresponding theory that represents primitive types, enumeration types or object types.

Undefined Values

In OCL, the evaluation of an expression may result in an undefined value, denoted by $\perp$. For example, an undefined value may result from querying the value of a property that has not been *set* to an object instance or from partially defined operations like division by zero. The general OCL rule is that, if one or more parts in an expression are undefined, then the complete expression will be undefined. Thus, all functions in our OCL type system remain partial, since any function can receive an undefined argument. The type for the undefined value $\perp$ is *OclVoid*, which is considered to be subtype of any other type in $\mathcal{M}$. Therefore, the $\perp$ value can be used as a value of any type. The algebraic semantics of the *OclVoid* type is represented in MEL by means of the kind concept in a natural way.

In the `OCL-COLLECTION-TYPES{T :: TRIV}` theory, undefined values are specified as terms that have no sort assigned to them, remaining in the kind of the corresponding sort. The *OclVoid* type is not represented by a specific sort in the `OCL-COLLECTION-TYPES{T :: TRIV}` theory. However, its semantics is provided by the equation

$$\boxed{[\![OclVoid]\!]_{MOF} = T_{\texttt{OCL-COLLECTION-TYPES\{T :: TRIV\}},[Collection+\{T\}]}}$$

6.1.3 The `OCL-DATATYPE-COLLECTIONS` theory

The `OCL-DATATYPE-COLLECTIONS` theory imports the theories `OCL-COLLECTIONS-TYPES{Bool}`, `OCL-COLLECTIONS-TYPES{String}`, `OCL-COLLECTIONS-TYPES{Int}`, `OCL-COLLECTIONS-TYPES{Float}` and `OCL-COLLECTIONS-TYPES{Oid}` so that their data elements are not manipulated by the inclusions

```
OCL-COLLECTIONS-TYPES{Bool},
OCL-COLLECTIONS-TYPES{String},
OCL-COLLECTIONS-TYPES{Int},
OCL-COLLECTIONS-TYPES{Float},
OCL-COLLECTIONS-TYPES{Oid}
⊆
OCL-DATATYPE-COLLECTIONS
```

The `OCL-DATATYPE-COLLECTIONS` theory permits the use of collection types of primitive type values in the algebraic representation of the metamodel realizations $reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ for the metamodel definition $\widetilde{\mathrm{MOF}}$, and $reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ for a metamodel definition $\widetilde{\mathcal{M}}$. The set of undefined values in the
`OCL-DATATYPE-COLLECTIONS` theory is defined by the equation:

$$\boxed{\begin{array}{ll} [\![OclVoid]\!]_{MOF} = & T_{\texttt{OCL-DATATYPE-COLLECTIONS},[Bool]} \cup \\ & T_{\texttt{OCL-DATATYPE-COLLECTIONS},[String]} \cup \\ & T_{\texttt{OCL-DATATYPE-COLLECTIONS},[Int]} \cup \\ & T_{\texttt{OCL-DATATYPE-COLLECTIONS},[Float]} \cup \\ & T_{\texttt{OCL-DATATYPE-COLLECTIONS},[Oid]} \end{array}}$$

6.1.4 The `MODEL` theory

The `MODEL` theory provides the algebraic representation for object types and model types in a metamodel definition $\widetilde{\mathcal{M}}$. This theory is parameterized with the `TH-OBJECT` theory, which is defined, in Maude notation, as

```
th TH-EOBJECT is
  including OCL-COLLECTIONS-TYPES{Oid} *
    (op empty-set to empty-set#Oid,
    op empty-orderedset to empty-orderedset#Oid,
    op empty-bag to empty-bag#Oid,
    op empty-sequence to empty-sequence#Oid) .

  sorts ObjectOid Cid Property PropertySet Object .
  subsort Property < PropertySet .
  subsort ObjectOid < Oid .

  op noneProperty : -> PropertySet .
  op _‘,_ : PropertySet PropertySet -> PropertySet
    [assoc comm id: noneProperty ctor] .
  op <_:_|_> : ObjectOid Cid PropertySet -> Object [obj ctor] .
  op nullObject : -> [Object] [ctor] .

  op oid : Object -> ObjectOid .
  op class : Object -> Cid .
  op getPropertySet : Object -> PropertySet .
endth
```

In this theory, object types are used to describe a model definition $\widetilde{M} : \mathcal{M}$ as a set of objects. Objects are defined using the following sorts: `ObjectOid` for object identifiers, where `ObjectOid < Oid`; `Cid` for class names; and `PropertySet` for multisets of comma-separated pairs of the form `(property : value)`, which represent property values. Objects in a model definition $\widetilde{M}$ are syntactically characterized by means of an operator

`<_:_|_> : ObjectOid Cid PropertySet -> Object`.

The operators `oid`, `class` and `getPropertySet` are defined to project the contents of an object, in Maude notation, as follows:

```
eq oid( < OID : CID | PS > ) = OID .
eq class( < OID : CID | PS > ) = CID .
eq getPropertySet( < OID : CID | PS > ) = PS .
```

where `OID : ObjectOid`, `CID : Cid`, and `PS : PropertySet`. Properties are defined as terms of sort `Property`, such that we have a subsort inclusion `Property < PropertySet`.

In the `MODEL{Obj :: TH-OBJECT}` theory, a model definition $\widetilde{M}$ that conforms to the metamodel $\mathcal{M}$, that is, such that $\widetilde{M} : \mathcal{M}$, can be viewed as a data element collection, which is represented as a set of objects by means of a `Configuration{OBJ}` term. A `Configuration{OBJ}` term is defined by means of the following constructors, in Maude notation:

```
op __ : ObjectCollection{OBJ} ObjectCollection{OBJ} ->
    ObjectCollection{OBJ} [ctor config assoc comm] .
op <<_>> : ObjectCollection{OBJ} -> Configuration{OBJ} [ctor] .
```

where the `Object` sort is a subsort of the `ObjectCollection{OBJ}` sort. That is, we first form a multiset of objects of sort `ObjectCollection{OBJ}` using the associative and commutative multiset union operator `_,_`, and then we *wrap* the set of objects using the `<<_>>` constructor to get the desired term of sort `Configuration{OBJ}`. The `MODEL{Obj :: TH-OBJECT}` theory is defined, in Maude notation, as

```
mod MODEL{OBJ :: TH-EOBJECT} is
    sorts ObjectCollection{OBJ} Configuration{OBJ} ModelType{OBJ}
        ConsistentModelType{OBJ} .
    subsorts OBJ$Object < ObjectCollection{OBJ} .
    subsorts ConsistentModelType{OBJ} < ModelType{OBJ} .
    subsorts ModelType{OBJ} < Configuration{OBJ} .

    op none : -> ObjectCollection{OBJ} [ctor] .
    op __ : ObjectCollection{OBJ} ObjectCollection{OBJ} ->
        ObjectCollection{OBJ}
        [assoc comm config id: none ctor format(d n d)] .

    op <<_>> : ObjectCollection{OBJ} -> Configuration{OBJ} [ctor] .
endm
```

In this theory, we have three sorts, namely `Configuration{OBJ}`, `ModelType{OBJ}`, and `ConsistentModelType{OBJ}`, that permit constraining the semantics of a model type, considering its values as sets of typed objects. In particular, the terms of sort `Configuration{OBJ}` represent collections of typed objects that may have property values. However, a model definition has a graph structure, by considering all the object-typed properties, which can also be viewed as a tree structure, by considering only the object-typed properties that are also defined as containment properties. This structure is not defined for terms of sort `Configuration{OBJ}`. A term $\widetilde{M}$ of the sort `Configuration{OBJ}` belongs `ModelType{OBJ}` iff it has a graph structure. This structure is checked by means of a membership axiom that assigns the sort `ModelType{OBJ}` to the term $\widetilde{M}$ if it has the proper graph structure. The structure of a model definition is developed in the following sections, by means of the $\mathit{reflect}_{\mathrm{MOF}}$ function. A model definition $\widetilde{M}$ belongs to the sort `ConsistentModelType{OBJ}` iff, in addition, satisfies a set of OCL constraints that are defined for the corresponding metamodel. The OCL constraint satisfaction relation is also defined as a decision procedure by means of a membership axiom in MEL. In Section 7, we define the OCL constraint satisfaction relation, indicating how the carrier of the sort `ConsistentModelType{OBJ}` is defined for a specific metamodel specification $(\mathcal{M}, \mathcal{C})$.

To instantiate the `MODEL{OBJ :: TH-OBJECT}` theory for a specific metamodel definition $\widetilde{\mathcal{M}}$, we define a view that maps the `TH-OBJECT` theory to another one that is generated from $\widetilde{\mathcal{M}}$ by means of the $\mathit{reflect}_{\mathrm{MOF}}$ function. We use the symbol $\mathcal{M}$ (the name of the root package of the metamodel definition $\widetilde{\mathcal{M}}$) as the view name. The $\mathit{reflect}_{\mathrm{MOF}}$ function is defined in the following sections. Therefore, when we instantiate the theory `MODEL{OBJ :: TH-OBJECT}` for a specific metamodel definition $\widetilde{\mathcal{M}}$ by means of the expression `MODEL{`$\mathcal{M}$`}`, we obtain the sorts `ModelType{`$\mathcal{M}$`}` and `ConsistentModelType{`$\mathcal{M}$`}`. We usually denote the sort `ModelType{`$\mathcal{M}$`}` by $\mathcal{M}$, and the sort `ConsistentModelType{`$\mathcal{M}$`}` by $(\mathcal{M}, \mathcal{C})$ for short.

The properties of an object can be accessed by using the dot notation, i.e., by means of an operator `_.property` that is provided in the `MODEL{OBJ :: TH-OBJECT}` theory for each property `property`. This operator is equationally defined in the expected way, so that its semantics in the initial algebra is that of a mapping

$$\texttt{\_.property : < OID : CID | property : value , PS >} \mapsto \texttt{value},$$

where `OID : Oid`, `CID : Cid`, `PS : PropertySet`, `property` is the name of the property, and `value` is the value of the corresponding property. For example, if we define an object term `c`, `c.property` obtains the value of the `property` meta-property. In addition, when `value` is a collection of object identifiers, we also define the function `_.property(_)`, which is defined by the equation

$$\texttt{< OID : CID | property : value , PS> . property}(\widetilde{\mathcal{M}}) = \{\widetilde{o} \in \widetilde{\mathcal{M}} \mid \texttt{oid}(\widetilde{o}) \in \texttt{value}\}.$$

The functions that project property values in an object are presented in detail in Section 7.

6.1.5 The `EXT-MODEL` theory

The $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory instantiates the `MODEL{OBJ :: TH-OBJECT}` theory with the view MOF, as explained in the following section, providing the algebraic semantics for the types that are defined in $\widetilde{\mathrm{MOF}}$. The `EXT-MODEL{OBJ :: TH-OBJECT}` theory includes both the `MODEL{OBJ}` theory and the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory, so that the type $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ can be used to define the type $[\![\mathcal{M}]\!]_{\mathrm{MOF}}$ in the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory. The theory `EXT-MODEL{OBJ :: TH-OBJECT}` is instantiated by the $\mathit{reflect}_{\mathrm{MOF}}$ function for a metamodel definition $\widetilde{\mathcal{M}}$ that is different from $\widetilde{\mathrm{MOF}}$, as shown in Section 6.3.

6.2 Algebraic Semantics of the MOF Metamodel

In this section, we describe the MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ that provides the type system and MEL axioms that can be used to define MOF metamodels. The $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory formalizes some of the concepts that appear in the MOF standard. More specifically, it formalizes the object types that are defined in a specific subset of the MOF metamodel definition: the *Essential MOF* metamodel definition, which we denote as $\widetilde{\mathrm{MOF}}$ from now on. Essential MOF (EMOF) [22] is

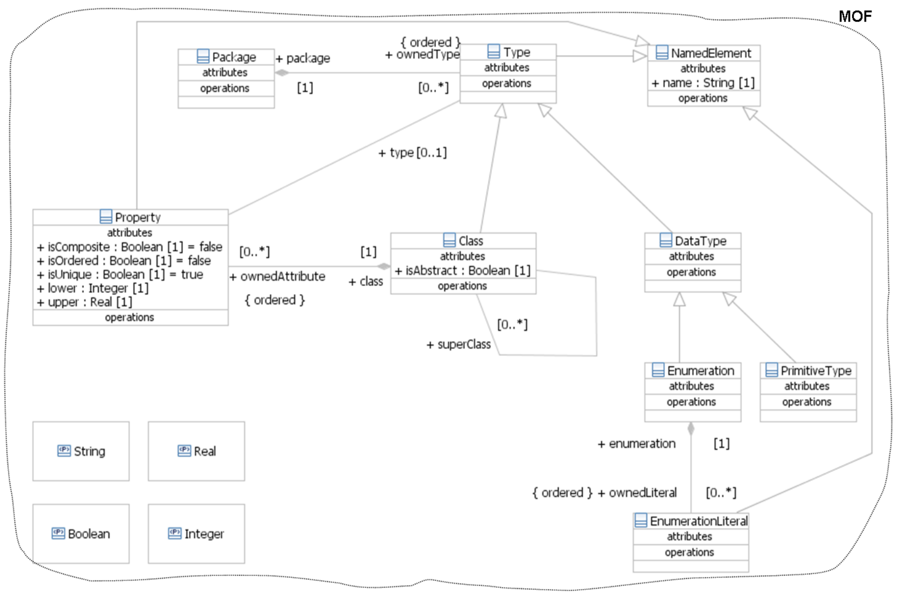


Figure 6.3: Simplification of the EMOF metamodel, in UML notation.

a subset of MOF that closely corresponds to the facilities found in object oriented programming languages and XML. A primary goal of EMOF is to provide a set of simple modeling concepts and to support class extension. EMOF is intended to enhance the combination of model-driven tool development and tool integration. We have chosen EMOF for its simplicity, and because it is a standard that is close to current modeling frameworks, such as the Eclipse Modeling Framework.

We show a simplification of the EMOF metamodel in Fig. 6.3, which includes the most relevant concepts that are used when a metamodel is defined. Since the EMOF metamodel reuses UML concepts to define the syntax of modeling languages, the UML notation is also reused for graphically representing the static structure of metamodels. Each concept is provided as an *object type* definition $\widetilde{\mathrm{OT}}$ in $\widetilde{\mathrm{MOF}}$, which is depicted as a class in Fig. 6.3. The object type definitions of the metamodel definition $\widetilde{\mathrm{MOF}}$ are related by means of an specialization relation $<_s$, represented as object type inheritance relationships in Fig. 6.3.

A metamodel definition $\widetilde{\mathcal{M}}$ has a graph structure where each object is a node of the graph and the object-typed properties are the edges, which are depicted as associations in Fig. 6.3. A metamodel definition $\widetilde{\mathcal{M}}$ can also be viewed as a tree by means of the containment properties that are defined between object types in $\widetilde{\mathrm{MOF}}$, which are depicted as composition aggregations in the figure. The $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory provides the $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ model type, whose values are metamodel definitions that can be viewed both as graphs, through object-typed properties, and trees, through containment relationships.

The $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory can be decomposed as

$$\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}) = \mathtt{MODEL}\{\mathtt{MOF}\} \cup \mathtt{MOFSTRUCTURE},$$

where `MOF` is a view that maps the sorts and operators of the `TH-OBJECT` theory to the elements of the `mod#MOF` theory, and the `MOFSTRUCTURE` theory provides the operators, equations and memberships that are needed to check the graph/tree structure of a specific metamodel definition $\widetilde{\mathcal{M}}$. The `mod#MOF` theory provides the operators that are specific to the metamodel definition $\widetilde{\mathrm{MOF}}$, which are needed to define objects in a metamodel definition $\widetilde{\mathcal{M}}$. The `mod#MOF` theory represents the actual parameter that instantiates the `MODEL{OBJ :: TH-OBJECT}` theory for the metamodel definition $\widetilde{\mathrm{MOF}}$.

In this section, we define the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory by providing: (a) the `mod#MOF` theory and the algebraic semantics of the object types that are defined in the metamodel definition $\widetilde{\mathrm{MOF}}$; (b) the algebraic semantics of the model types that are provided by the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory; (c) the algebraic semantics of the $[\![\text{MOF}]\!]_{\text{MOF}}$ model type by means of a membership axiom, which is defined in the `MOFSTRUCTURE` theory; and (d) the graphical representation of a metamodel definition $\widetilde{\mathcal{M}}$.

6.2.1 Algebraic Semantics of MOF Object Types

Object types are used to describe a metamodel definition $\widetilde{\mathcal{M}}$: MOF as a collection of objects. In the `mod#MOF` theory, objects are defined using the following sorts: `Oid#MOF` for object identifiers, where `Oid#MOF < Oid`; `Cid#MOF` for class names; and `PropertySet#MOF` for multisets of comma-separated pairs of the form `(property : value)`, which represent property values. Objects in a metamodel definition $\widetilde{\mathcal{M}}$ are then syntactically characterized in the theory $\mathit{reflect}_{\text{MOF}}(\widetilde{\text{MOF}})$ by means of an operator

`<_:_|_> : Oid#MOF Cid#MOF PropertySet#MOF -> Object#MOF`.

A constant

```
op nullObject#MOF : -> [Object#MOF] [ctor] .
```

permits defining undefined values for objects. Three operators are defined to project the contents of an object, which are defined, in Maude notation, as follows:

```
op oid : Object#MOF -> Oid#MOF .
eq oid( < OID : CID | PS > ) = OID .

op class : Object#MOF -> Cid#MOF .
eq class( < OID : CID | PS > ) = CID .

op propertySet : Object#MOF -> Cid#MOF .
eq propertySet( < OID : CID | PS > ) = PS .
```

where `OID : Oid#MOF`, `CID : Cid#MOF`, and `PS : PropertySet#MOF`. Properties are defined as terms of a sort `Property#MOF`, such that we have a subsort inclusion `Property#MOF < PropertySet#MOF`. The constant

```
nullProperty#MOF : -> PropertySet#MOF
```

permits defining an object without property values.

The view `MOF` that maps the `TH-OBJECT` to the `mod#MOF` theory is defined, in Maude notation, as follows:

```
view MOF from TH-EOBJECT to mod#MOF is
  sort Cid to Cid#MOF .
  sort Object to Object#MOF .
  sort ObjectOid to Oid#MOF .
  sort Property to Property#MOF .
  sort PropertySet to PropertySet#MOF .
  op noneProperty to noneProperty#MOF .
  op nullEObject to nullObject#MOF .
endv
```

Therefore, the `MODEL{MOF}` theory instantiates the theory `MODEL{OBJ :: TH-OBJECT}` with the `MOF` view, providing the types that are needed to represent collections of typed objects as terms of sort `Configuration{MOF}`, also called MOF_0. Note that a `Configuration{MOF}` term is not a metamodel definition yet. This term becomes a metamodel definition $\widetilde{\mathcal{M}}$ when it belongs to the carrier of the sort `ModelType{MOF}`, also called MOF. This is achieved by means of the membership axiom that is defined in the `MOFSTRUCTURE` theory, defined below.

The `mod#MOF` theory also includes the `OCL-DATATYPE-COLLECTIONS` theory, so that any of the sorts that are defined in this theory, primitive types and OCL collection types, can be used to define operator arguments.

We explain the syntactic representation of the object types that are defined in $\widetilde{\text{MOF}}$ in the MEL theory `mod#MOF` so that they can be used to define metamodel definitions $\widetilde{\mathcal{M}}$ in the model definition `MODEL{MOF}` theory[2]. To illustrate how the types of the theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ can be used in a metamodel definition $\widetilde{\mathcal{M}}$, we have taken the metamodel definition $\widetilde{\mathrm{MOF}}$ itself as an example. This syntactic algebraic representation is generalized for any MOF metamodel $\widetilde{\mathcal{M}} : \mathrm{MOF}$ in Section 6.3. The object type definitions $\widetilde{\mathrm{OT}}$ that we have considered in the metamodel definition $\widetilde{\mathrm{MOF}}$ are the following:

NamedElement

The abstract object type NamedElement provides the attribute *name* and is syntactically represented in the MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$, in Maude notation, as follows:

```
sorts NamedElement oid#NamedElement .
subsort NamedElement < Cid#MOF .
subsort oid#NamedElement < Oid#MOF .
op name‘:_ : String -> Property#MOF .
op name : -> Property#MOF .
```

The semantics of the object type NamedElement in the initial algebra of the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory is defined by the equation

$$\boxed{\begin{array}{l}[\![\textsc{NamedElement}]\!]_{\mathrm{MOF}} = \\ \{\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{MOF}),\mathit{Object\#MOF}} \mid \mathit{class}(\widetilde{o}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{MOF}),\mathit{NamedElement}}\},\end{array}}$$

and the *instanceOf* relation between an object $\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{MOF}),\mathit{Object\#MOF}}$ and the object type NamedElement is defined by the following equivalence:

$$\boxed{\widetilde{o} : \textsc{NamedElement} \;\Leftrightarrow\; \widetilde{o} \in [\![\textsc{NamedElement}]\!]_{\mathrm{MOF}}}$$

Type

The abstract object type Type defines the common structure of objects that can participate as object property types in object types of a metamodel definition $\widetilde{\mathcal{M}} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\mathit{Configuration}\{\mathrm{MOF}\}}$. It provides the meta-property *package*, indicating the Package instance that contains it. The abstract object type Type is specialized by both the DataType and the Class object types, so that any type can be enclosed in a Package instance. This class is syntactically represented, in Maude notation, in the MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ as follows:

```
sorts Type oid#Type .
subsort Type < NamedElement .
subsort oid#Type < oid#NamedElement .
op package‘:_ : Oid -> Property#MOF .
op package : -> Property#MOF .
```

The semantics of the object type Type in the initial algebra of the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory is defined by the equation

$$\boxed{\begin{array}{l}[\![\textsc{Type}]\!]_{\mathrm{MOF}} = \\ \{\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{MOF}),\mathit{Object\#MOF}} \mid \mathit{class}(\widetilde{o}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{MOF}),\mathit{Type}}\},\end{array}}$$

and the *instanceOf* relation between an object $\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{MOF}),\mathit{Object\#MOF}}$ and the object type Type is defined by the following equivalence:

$$\boxed{\widetilde{o} : \textsc{Type} \;\Leftrightarrow\; \widetilde{o} \in [\![\textsc{Type}]\!]_{\mathrm{MOF}}}$$

[2]This syntactic representation will be given in the usual algebraic manner by specifying a collection of sorts (algebraic types) and subsorts (subtypes), and certain operator declarations, so that each algebraic term can be built up using such operators. Note that this is entirely analogous to using a context-free grammar to specifying the algebraic terms, with nonterminals corresponding to sorts, and operator declarations corresponding to grammar rules.

Class

Object types are the central concept of EMOF to model entities of the problem domain in metamodels. An object type OT is defined in $\widetilde{MOF}$ as a Class instance $\widetilde{c}$ and a set of Property instances $\widetilde{p}$. Therefore, a user can define new object types in the form of Class instances and Property instances. The object type Class contains the following meta-properties: *name*, indicates the name of the object type definition $\widetilde{\mathrm{OT}}$; *abstract*, indicates whether the object type can be instantiated (*abstract = false*) or not (*abstract = true*); and *superClass*, indicates that the object type is defined as a specialization of the object types that are referred to by means of this property. The constructors that permit defining objects of the object type Class are defined, in Maude notation, as follows:

```
sort Class oid#Class .
subsort Class < Type .
subsort oid#Class < oid#Type .
op Class : -> Class .
op oid#Class : Qid -> oid#Class .
op isAbstract`:_ : Bool -> Property#MOF .
op isAbstract : -> Property#MOF .
op superClass`:_ : OrderedSet{Oid} -> Property#MOF
op superClass : -> Property#MOF
op ownedAttribute`:_ : OrderedSet{Oid} -> Property#MOF .
op ownedAtribute :  -> Property#MOF .
```

The Class instance that defines the name of the object type Class in the metamodel MOF is defined as the term

```
< oid#Class('Class0) : Class |
    name : "Class", isAbstract : false,
    package : oid#Package('Package0)
>.
```

The semantics of the object type Class in the initial algebra of the $reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory is defined by the equation

$$\boxed{[\![\mathrm{Class}]\!]_{\mathrm{MOF}} = \{\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{MOF}),Object\#MOF} \mid class(\widetilde{o}) \in T_{reflect_{\mathrm{MOF}}(\widetilde{MOF}),Class}\},}$$

and the *instanceOf* relation between an object $\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{MOF}),Object\#MOF}$ and the object type Class is defined by the following equivalence:

$$\boxed{\widetilde{o} : \mathrm{Class} \quad \Leftrightarrow \quad \widetilde{o} \in [\![\mathrm{Class}]\!]_{\mathrm{MOF}}}$$

Property

The object type Property permits describing attributes of object type definitions in $\widetilde{\mathcal{M}}$, or relationships among object types. The object type Property is defined in the meta-metamodel definition $\widetilde{\mathrm{MOF}}$ with the following metaproperties: *name*, indicates the name of the property; *lower*, indicates the minimum multiplicity of the property, whose values can be *1* if a value is required for the property, or *0* if no value is required; *upper*, indicates the maximum multiplicity of the property, whose values can be *1* if a value of the property is not a collection, or *-1* (usually written *) if a value of the property is a collection[3]; *ordered*, indicates that a collection of values must be ordered for the property, when *upper = -1*; *unique*, indicates that a collection of values for the property cannot contain duplicates, when *upper = -1*; *opposite*, indicates that an object type $\widetilde{\mathrm{OT2}}$, which is used to type a property instance $\widetilde{p1}$ in an object type $\widetilde{\mathrm{OT1}}$, has a property $\widetilde{p2}$, which is typed with the object type $\widetilde{\mathrm{OT1}}$; *class*, indicates the class instance $\widetilde{c}$ of the object type $\widetilde{\mathrm{OT}}$ that owns the property; *type*, defines the type of the property, where the type can be a basic type definition, an

[3]We do not consider values n, such that $1 < n$, for the upper metaproperty.

enumeration type definition or an object type definition; and *isComposite*, indicates that a property, whose *class* meta-property refers to a class instance $\widetilde{cl1}$ of an object type $\widetilde{\mathrm{OT1}}$ and whose *type* meta-property refers to a class instance $\widetilde{cl2}$ of an object type $\widetilde{\mathrm{OT2}}$, defines a composition relationship between object types $\widetilde{\mathrm{OT1}}$ and $\widetilde{\mathrm{OT2}}$, where $\widetilde{\mathrm{OT2}}$ is the composite object type and $\widetilde{\mathrm{OT1}}$ is the component object type; and *defaultValue*, indicates a literal that represents the default value of the property when the property is not initialized. The meta-properties *lower*, *upper*, *ordered* and *unique* constitute the multiplicity metadata of a specific property [4].

The constructors that permit defining objects of the object type PROPERTY are defined, in Maude notation, as follows:

```
sort Property oid#Property .
subsort Property < NamedElement .
subsort oid#Property < oid#NamedElement .
op Property : -> Property .
op oid#Property : Qid -> oid#Property .
op lower‘:_ : Int -> Property#MOF .
op lower : -> Property#MOF .
op upper‘:_ : Int -> Property#MOF .
op upper : -> Property#MOF .
op isOrdered‘:_ : Bool -> Property#MOF .
op isOrdered : -> Property#MOF .
op isUnique‘:_ : Bool -> Property#MOF .
op isUnique : -> Property#MOF .
op isComposite‘:_ : Bool -> Property#MOF .
op isComposite : -> Property#MOF .
op opposite‘:_ : [Oid] -> Property#MOF .
op opposite : -> Property#MOF .
op class‘:_ : Oid -> Property#MOF .
op class : -> Property#MOF .
op defaultValue‘:_ : String -> Property#MOF .
op defaultValue : -> Property#MOF .
```

The PROPERTY instance $\widetilde{p}$ that defines the metaproperty name of the object type CLASS in $\widetilde{\mathrm{MOF}}$ is represented by the term

```
< oid#Property(’Property0) : Property |
    name : "name", lower : 1, upper: 1,
    isOrdered, isUnique, isComposite,
    type : oid#PrimitiveType(’PrimitiveType0),
    class : oid#Class(’Class0)
>.
```

In EMOF, an association between two classes can be defined by means of two properties that are defined as *opposite*, i.e., each one refers to the other by means of the *opposite* association end. For example, the composition that is defined between the classes `Enumeration` and `EnumerationLiteral` by means of the `ownedLiteral` and `enumeration` opposite references is represented as follows:

```
< oid#Class(’class0) : Class |
    name : "Enumeration",
    ownedAttribute : OrderedSet{oid#Property(’prop0)}
>
< oid#Property(’prop0) : Property |
    name : "ownedLiteral",
    class : oid#Class(class0),
    opposite : oid#Property(’prop1)
>
< oid#Class(’class1) : Class |
```

[4]In the metamodel of the MOF 2.0 specification, multiplicity meta-properties belong to the *MultiplicityElement* object type.

```
    name : "EnumerationLiteral",
    ownedAttribute : OrderedSet{oid#Property('prop1)}
>
< oid#Property('prop1) : Property |
    name : "enumeration",
    isComposite : true,
    class : oid#Class('class1),
    opposite : oid#Property('prop0)
>.
```

The semantics of the object type PROPERTY in the initial algebra of the $reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory is defined by the equation

$$\boxed{[\![\text{PROPERTY}]\!]_{\mathrm{MOF}} = \{\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{MOF}), Object\#MOF} \mid class(\widetilde{o}) \in T_{reflect_{\mathrm{MOF}}(\widetilde{MOF}), Property}\},}$$

and the *instanceOf* relation between an object $\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{MOF}), Object\#MOF}$ and the object type PROPERTY is defined by the following equivalence:

$$\boxed{\widetilde{o} : \text{PROPERTY} \;\Leftrightarrow\; \widetilde{o} \in [\![\text{PROPERTY}]\!]_{\mathrm{MOF}}}$$

DataType

The object type DATATYPE describes any type that does not constitute an object type definition $\widetilde{\mathrm{OT}}$, i.e., types whose values do not change over time. For example, the integer 1 always represents the same value. The INTEGER type is defined as an instance of the DATATYPE object type. The object type DATATYPE is specialized by both the PRIMITIVETYPE and the ENUMERATION object types. The object type DATATYPE is abstract and is represented in the $reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ by the following sorts and constructors:

```
sort DataType oid#DataType .
subsort DataType < Type .
subsort oid#DataType < oid#Type .
```

The semantics of the object type DATATYPE in the initial algebra of the $reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory is defined by the equation

$$\boxed{[\![\text{DATATYPE}]\!]_{\mathrm{MOF}} = \{\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{MOF}), Object\#MOF} \mid class(\widetilde{o}) \in T_{reflect_{\mathrm{MOF}}(\widetilde{MOF}), DataType}\},}$$

and the *instanceOf* relation between an object $\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{MOF}), Object\#MOF}$ and the object type DATATYPE is defined by the following equivalence:

$$\boxed{\widetilde{o} : \text{DATATYPE} \;\Leftrightarrow\; \widetilde{o} \in [\![\text{DATATYPE}]\!]_{\mathrm{MOF}}}$$

PrimitiveType

The object type PRIMITIVETYPE permits the definition of basic data types that can be used when a model $\widetilde{M} : \mathrm{MOF}$ is defined. For example, we take into account four instances of the object type PRIMITIVETYPE in $\widetilde{\mathrm{MOF}}$: $\widetilde{\text{BOOLEAN}}$, $\widetilde{\text{STRING}}$, $\widetilde{\text{INTEGER}}$ and $\widetilde{\text{REAL}}$[5]. The constructors that permit defining objects of the object type PRIMITIVETYPE are defined, in Maude notation, as follows:

[5]In the EMOF specification, the primitive type *UnlimitedNatural* is provided instead of Real. We have added this modification to the metamodel to align it with the OCL type system.

```
sort PrimitiveType oid#PrimitiveType .
subsort PrimitiveType < DataType .
subsort oid#PrimitiveType < oid#Type .
op PrimitiveType : -> PrimitiveType .
op oid#PrimitiveType : Qid -> oid#PrimitiveType .
```

The PrimitiveType instance $\widetilde{\text{String}}$ that defines the String data type is represented by means of the term

```
< oid#PrimitiveType('PrimitiveType0) : PrimitiveType |
    name : "String", package : oid#Package('Package0) > .
```

The semantics of the object type PrimitiveType in the initial algebra of the $\mathit{reflect}_{\text{MOF}}(\widetilde{\text{MOF}})$ theory is defined by the equation

$$\boxed{\begin{array}{l} [\![\text{PrimitiveType}]\!]_{\text{MOF}} = \\ \\ \{\widetilde{o} \in T_{\mathit{reflect}_{\text{MOF}}(\widetilde{MOF}),Object\#MOF} \mid \mathit{class}(\widetilde{o}) \in T_{\mathit{reflect}_{\text{MOF}}(\widetilde{MOF}),PrimitiveType}\}, \end{array}}$$

and the *instanceOf* relation between an object $\widetilde{o} \in T_{\mathit{reflect}_{\text{MOF}}(\widetilde{MOF}),Object\#MOF}$ and the object type PrimitiveType is defined by the following equivalence:

$$\boxed{\widetilde{o} : \text{PrimitiveType} \quad \Leftrightarrow \quad \widetilde{o} \in [\![\text{PrimitiveType}]\!]_{\text{MOF}}}$$

Enumeration and EnumerationLiteral

An enumeration type is a finite set of literal values. It can be defined by means of an instance of the object type Enumeration and a set of instances of the object type EnumerationLiteral. The constructors that permit defining instances of the object types Enumeration and EnumerationLiteral are represented, in Maude notation, as follows:

```
sorts Enumeration oid#Enumeration
    EnumerationLiteral oid#EnumerationLiteral .
subsort Enumeration < DataType .
subsort oid#Enumeration < oid#DataType .
subsort EnumerationLiteral < NamedElement .
subsort oid#EnumerationLiteral < oid#NamedElement .
op ownedLiteral`:_ : OrderedSet{Oid} -> Property#MOF .
op ownedLiteral :  -> Property#MOF .
op enumeration`:_ : Oid -> Property#MOF .
op enumeration :  -> Property#MOF .
```

As an example, we take the enumeration that is defined in the metamodel definition $\widetilde{\text{RDBMS}}$. This enumeration is represented as the following collection of objects:

```
< oid#Enumeration('Enum0) : Enumeration |
    name : "RDataType", literal :
    OrderedSet{
        oid#EnumerationLiteral('Literal0) ::
        oid#EnumerationLiteral('Literal1) ::
        oid#EnumerationLiteral('Literal2) ::
        oid#EnumerationLiteral('Literal3) ::
        oid#EnumerationLiteral('Literal4)
    },
    package : oid#Package('Package0)
>
< oid#EnumerationLiteral('Literal0) : EnumerationLiteral |
    name : "VARCHAR", enumeration : oid#Enumeration('Enum0) >
```

```
< oid#EnumerationLiteral('Literal1) : EnumerationLiteral |
    name : "NUMBER", enumeration : oid#Enumeration('Enum0) >
< oid#EnumerationLiteral('Literal2) : EnumerationLiteral |
    name : "BOOLEAN", enumeration : oid#Enumeration('Enum0) >
< oid#EnumerationLiteral('Literal3) : EnumerationLiteral |
    name : "DATE", enumeration : oid#Enumeration('Enum0) >
< oid#EnumerationLiteral('Literal4) : EnumerationLiteral |
    name : "DECIMAL", enumeration : oid#Enumeration('Enum0) > .
```

The semantics of the object type ENUMERATION in the initial algebra of the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory is defined by the equation

$$\boxed{\begin{array}{l}[\![\textsc{Enumeration}]\!]_{\mathrm{MOF}} = \\ \\ \{\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{MOF}),\mathit{Object\#MOF}} \mid \mathit{class}(\widetilde{o}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{MOF}),\mathit{Enumeration}}\},\end{array}}$$

and the *instanceOf* relation between an object $\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{MOF}),\mathit{Object\#MOF}}$ and the object type ENUMERATION is defined by the following equivalence:

$$\boxed{\widetilde{o} : \textsc{Enumeration} \quad \Leftrightarrow \quad \widetilde{o} \in [\![\textsc{Enumeration}]\!]_{\mathrm{MOF}}}$$

The semantics of the object type ENUMERATIONLITERAL in the initial algebra of the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory is defined by the equation

$$\boxed{\begin{array}{l}[\![\textsc{EnumerationLiteral}]\!]_{\mathrm{MOF}} = \\ \\ \quad \{\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{MOF}),\mathit{Object\#MOF}} \mid \\ \\ \quad \mathit{class}(\widetilde{o}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{MOF}),\mathit{EnumerationLiteral}}\},\end{array}}$$

and the *instanceOf* relation between an object $\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{MOF}),\mathit{Object\#MOF}}$ and the object type ENUMERATIONLITERAL is defined by the following equivalence:

$$\boxed{\widetilde{o} : \textsc{EnumerationLiteral} \quad \Leftrightarrow \quad \widetilde{o} \in [\![\textsc{EnumerationLiteral}]\!]_{\mathrm{MOF}}}$$

Package

A PACKAGE instance is the root element of an EMOF model and encapsulates all user-defined types that describe a metamodel, i.e., instances of the object type TYPE. In EMOF, a PACKAGE instance can also contain other PACKAGE instances by means of the *nestedPackage* meta-property. The *nestingPackage* meta-property indicates its container package, if any. The constructors that permit defining objects of the object type PACKAGE are defined, in Maude notation, as follows:

```
sort Package oid#Package .
subsort Package < NamedElement .
subsort oid#Package < oid#NamedElement .
op Package : -> Package .
op oid#Package : Qid -> oid#Package .
op ownedType`:_ : OrderedSet{Oid} -> Property#MOF .
op ownedType : -> Property#MOF .
op nestedPackage`:_ : OrderedSet{Oid} -> Property#MOF .
op nestedPackage : -> Property#MOF .
op nestingPackage`:_ : [Oid] -> Property#MOF .
op nestingPackage : -> Property#MOF .
```

The PACKAGE instance $\widetilde{pk}$ that encapsulates the types that are defined in the metamodel definition $\widetilde{\mathrm{MOF}}$ is represented by means of the term

```
< oid#Package('Package0) : Package |
    name : "EMOF",
    nestedPackage, nestingPackage,
    ownedType : OrderedSet{
        oid#Class('Class0) ::
        oid#PrimitiveType('PrimitiveType0) :: ...
    }
>
```

6.2.2 Algebraic Semantics of MOF Model Types

The $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory provides the following model types:

(i) MOF_0, which is represented by the sort $\mathit{Configuration}\{\mathrm{MOF}\}$ in the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory, is the type of collections of typed objects. The semantics of the MOF_0 type is defined by the equation

$$\boxed{[\![\mathrm{MOF}_0]\!]_{\mathrm{MOF}} = T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\mathit{Configuration}\{MOF\}},}$$

where

$$\boxed{\widetilde{\mathcal{M}} : \mathrm{MOF}_0 \Leftrightarrow \widetilde{\mathcal{M}} \in [\![\mathrm{MOF}_0]\!]_{\mathrm{MOF}}.}$$

(ii) MOF, which is represented by $\mathit{ModelType}\{MOF\}$ in the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory, is the type of collections of typed objects that keep both a graph and a tree structure, constituting what we call a metamodel definition $\widetilde{\mathcal{M}}$. The semantics of the MOF type is defined by the equation

$$\boxed{[\![\mathrm{MOF}]\!]_{\mathrm{MOF}} = T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\mathit{ModelType}\{MOF\}},}$$

where

$$\boxed{\widetilde{\mathcal{M}} : \mathrm{MOF} \Leftrightarrow \widetilde{\mathcal{M}} \in [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}.}$$

There is no constructor in the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory to define terms of sort `ModelType{MOF}`. The domain $T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\mathit{ModelType}\{MOF\}}$ is defined by means of the conditional membership that is defined in the `MOFSTRUCTURE` theory, which is presented below.

(iii) $(\mathrm{MOF},\mathcal{C}_{MOF})$, which is represented by $\mathit{ConsistentModelType}\{MOF\}$ in the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory, is the type of metamodel definitions that both conform to MOF and satisfy the OCL constraints $\mathcal{C}_{\mathrm{MOF}}$. The semantics of the $(\mathrm{MOF},\mathcal{C}_{MOF})$ type is defined by the equation

$$\boxed{[\![(\mathrm{MOF},\mathcal{C}_{MOF})]\!]_{\mathrm{MOF}} = T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\mathit{ConsistentModelType}\{MOF\}},}$$

where

$$\boxed{\widetilde{\mathcal{M}} : (\mathrm{MOF},\mathcal{C}_{MOF}) \Leftrightarrow \widetilde{\mathcal{M}} \in [\![(\mathrm{MOF},\mathcal{C}_{MOF})]\!]_{\mathrm{MOF}}.}$$

The domain $T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\mathit{ConsistentModelType}\{MOF\}}$ is defined in Section 7.

6.2.3 Structure of a Metamodel Definition

A metamodel definition $\widetilde{\mathcal{M}}$, $\widetilde{\mathcal{M}} : \mathrm{MOF}_0$, is defined as a collection of typed objects. In an object $\widetilde{o}$, such that $\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\mathit{Object\#MOF}}$, an *object-typed property (or reference)* is an expression of the form *(prop : value)* or *prop*, such that $(\mathit{prop} : \mathit{value}),(\mathit{prop}) \in \mathit{getProperties}(\widetilde{o})$, which is defined in $\widetilde{\mathrm{MOF}}$ by means of a Property instance $\widetilde{p}$, such that $\widetilde{p}.\mathit{type}(\widetilde{\mathrm{MOF}}) : \textsc{Class}$. A *value-typed property (or attribute)* is an expression of the form *(prop : value)* or *prop*, such that $(\mathit{prop} : \mathit{value}),(\mathit{prop}) \in \mathit{getProperties}(\widetilde{o})$, which is defined in $\widetilde{\mathrm{MOF}}$ by means of a Property instance $\widetilde{p}$, such that $\widetilde{p}.\mathit{type}(\widetilde{\mathrm{MOF}}) : \textsc{DataType}$. A property value *(prop*

: value) represents a property that has been initialized with a certain *value*, and *prop* represents an unset property that has not been initialized yet.

A metamodel definition $\widetilde{\mathcal{M}}$ can be viewed as a graph, by considering the object-typed properties that are defined in $\widetilde{\text{MOF}}$, or as a tree, by considering the containment properties that are defined in $\widetilde{\text{MOF}}$. This structure is preserved in the algebraic semantics of the data type $[\![\text{MOF}]\!]_{\text{MOF}}$ by means of a membership axiom, which is defined in the theory `MOFSTRUCTURE`. In subsequent paragraphs, we define this membership axiom in MEL, which permits defining types in metamodel definitions and the specialization relation between object types. We define the $[\![\text{MOF}]\!]_{\text{MOF}}$ type by constraining the semantics of the $[\![\text{MOF}_0]\!]_{\text{MOF}}$ type gradually: (i) giving the graph structure of a metamodel definition $\widetilde{\mathcal{M}}$, (ii) giving the tree structure of a metamodel definition $\widetilde{\mathcal{M}}$, and (iii) giving some additional constraints about well-formed metamodel definitions. In Section 8, we indicate how this membership axiom can be redefined using the MOF Reflection facilities support.

Graph structure

A collection of typed objects $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF, can be viewed as a graph where objects are nodes in the graph, and object-typed properties define edges in the graph. We consider part of the metamodel definition $\widetilde{\text{MOF}}$, which is represented as the term[6]

```
<<
    < oid#Package('pk1) : Package |
        ownedAttribute : OrderedSet{
            oid#Class('cl1) ::
            oid#Class('cl2) ::
            oid#Class('cl3) ::
            oid#Class('cl4) ::
            oid#PrimtiveType('t1) ::
            oid#PrimtiveType('t2)
        }
    >
    < oid#Class('cl1) : Class |
        name : "NamedElement"
    >
    < oid#Class('cl2) : Class |
        name : "Type"
    >
    < oid#Class('cl3) : Class |
        name : "Class"
    >
    < oid#Class('cl4) : Class |
        name : "Property"
    >
    < oid#PrimtiveType('t1) : PrimitiveType |
        name : "String"
    >
    < oid#PrimtiveType('t2) : PrimitiveType |
        name : "Boolean"
    >
    < oid#Property('p11) : Property |
        name : "name"
        type : oid#PrimtiveType('t1)
    >
    < oid#Property('p31) : Property |
        name : "isAbstract"
        type : oid#PrimtiveType('t2)
    >
```

[6]We have simplified the term by only adding the properties that we want to consider in the example. A detailed representation of a metamodel definition as term can be found in Appendix B.

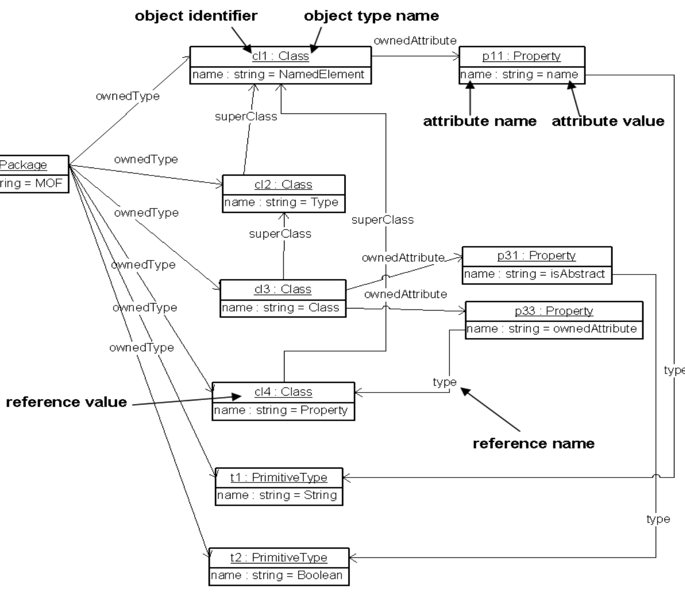


Figure 6.4: Graphical representation of part of the metamodel definition $\widetilde{\text{MOF}}$ as a graph.

```
    < oid#Property('p32) : Property |
        name : "ownedAttribute"
        type : oid#Class('cl4)
    >
>>.
```

This term can be represented as a graph, as shown in Fig. 6.4. In the figure, each node of the graph is represented by a square that is split in two parts. Each node is an object whose identifier[7] and object type name are shown in the first part of the square, and whose value-typed properties are shown in the second part of the square. In the figure, each edge represents an object-typed property value.

An edge in the graph of a metamodel definition $\widetilde{\mathcal{M}}$ is represented as a pair $(\widetilde{o_1}, \widetilde{o_2})$, where $\widetilde{o_1}, \widetilde{o_2} \in T_{\mathit{reflect}_{\text{MOF}}(\widetilde{\text{MOF}}),\mathit{Object\#MOF}}$, $\widetilde{o_1}, \widetilde{o_2} \in \widetilde{\mathcal{M}}$, $\widetilde{o_1}$ is the object that contains the object-typed property, and $\widetilde{o_2}$ is the object that is referred to by means of the property. Given a collection of typed objects $\widetilde{\mathcal{M}}$, the collection of edges that are defined in $\widetilde{\mathcal{M}}$ is given by the partial function

$$\begin{aligned} \mathit{edges} :\ & [\![\text{MOF}]\!]_{\text{MOF}} \times [\![\text{MOF}]\!]_{\text{MOF}} \rightsquigarrow \\ & \mathcal{P}_{\mathit{fin}}(T_{\mathit{reflect}_{\text{MOF}}(\widetilde{\text{MOF}}),\mathit{Object\#MOF}} \times T_{\mathit{reflect}_{\text{MOF}}(\widetilde{\text{MOF}}),\mathit{Object\#MOF}}), \end{aligned}$$

which is only defined, by now, for pairs of the form $(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}})$, where: $\widetilde{\mathcal{M}} : \text{MOF}$, and $\widetilde{\text{MOF}}$, such that $\widetilde{\text{MOF}} : \text{MOF}$, is the definition of the MOF meta-metamodel. In this case, the *edges* function is defined by the mapping:

$$\begin{aligned} \mathit{edges}(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}}) = \quad \{\ (\widetilde{o_1}, \widetilde{o_2})\ |\quad & \widetilde{o_1}, \widetilde{o_2} \in T_{\mathit{reflect}_{\text{MOF}}(\widetilde{\text{MOF}}),\mathit{Object\#MOF}} \ \wedge\ \widetilde{o_1}, \widetilde{o_2} \in \widetilde{\mathcal{M}}\ \wedge \\ & \exists \widetilde{p}\ (\widetilde{p} : \textsc{Property}\ \wedge\ \widetilde{p} \in \widetilde{\text{MOF}}\ \wedge \\ & \quad \widetilde{p}.\mathit{type}(\widetilde{\text{MOF}}) : \textsc{Class}\ \wedge \\ & \quad \widetilde{o_2} \in \widetilde{o_1}.(\widetilde{p}.\mathit{name})(\widetilde{\mathcal{M}})) \\ \}. \end{aligned}$$

[7]In the figure, an object identifier of the form `oid#Class('cl1)` is simplified as `cl1`.

Definition 1 (MOF Graph)**.** *Given a collection $\widetilde{\mathcal{M}}$ of typed objects, such that $\widetilde{\mathcal{M}} : \mathrm{MOF}$, and the definition of the meta-metamodel $\widetilde{\mathrm{MOF}}$, $graph(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$ is called the* MOF graph *of the metamodel definition $\widetilde{\mathcal{M}}$ and is defined by the equation*

$$graph(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = (V_{\mathcal{M}}, E_{\mathcal{M}}),$$

where:

(i) $V_{\mathcal{M}}$ is a collection of typed objects that constitutes the set of nodes of the graph, and is defined by the equation $V_{\mathcal{M}} = \widetilde{\mathcal{M}}$; and

(ii) $E_{\mathcal{M}}$ is the set of edges of the graph, and is defined by the equation

$$E_{\mathcal{M}} = edges(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}).$$

SpecMEL is the data type of *finitely-presented* MEL theories, that is, theories of the form $(S, <, \Omega, E \cup A)$, where all the components are finite. Without loss of generality we assume countable sets *Sorts* and *OpNames,* so that each set of sorts S is a finite subset of *Sorts*, and the operator names in Ω are a finite subset of *OpNames*. To obtain the sort that corresponds to a class, we define the function $ClassSort : \textsc{Class} \rightarrow Sorts$, which is defined for MOF Class instances as follows: $ClassSort : \widetilde{cl} \mapsto \widetilde{cl}.name$, where $(\widetilde{cl} : \textsc{Class})$.

The data type $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ is defined, in a first step, by the set of collections of typed objects that are graphs. To denote that a collection of typed objects $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}} : \mathrm{MOF}_0$, is a MOF graph, we can also use the expression $\widetilde{\mathcal{M}} : \mathrm{MOF}$ by taking into account the following equivalence

$$\widetilde{\mathcal{M}} : \mathrm{MOF} \iff graph(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) \text{ is a MOF } graph.$$

The values of the type $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ are collections of typed objects that keep a graph structure. This structure is checked over a collection of typed objects $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}} : \mathrm{MOF}_0$, by means of the following conditional membership:

$$\left.\begin{array}{l} \widetilde{\mathcal{M}} : \mathrm{MOF}_0 \ \wedge \\[1ex] noBrokenLinks(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = true \ \wedge \\[1ex] wellTypedLinks(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = true \end{array}\right\} \Longrightarrow \widetilde{\mathcal{M}} : \mathrm{MOF}$$

where *noBrokenLinks* is a function

$$noBrokenLinks : [\![\mathrm{MOF}_0]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow [\![\textsc{Boolean}]\!]_{\mathrm{MOF}},$$

which checks that there are no broken edges in the graph $graph(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$, i.e., an object in $\widetilde{\mathcal{M}}$ does not refer to an undefined object by means of an object-typed property. This function is defined as follows:

$$\begin{array}{l}
noBrokenLinks(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = false \\
\quad when\ \exists \widetilde{o_1}(\widetilde{o_1} \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), Object\#MOF} \ \wedge\ \widetilde{o_1} \in \widetilde{\mathcal{M}} \ \wedge \\[1ex]
\qquad \exists \widetilde{p}\ (\widetilde{p} : \textsc{Property} \ \wedge\ \widetilde{p} \in \widetilde{\mathrm{MOF}} \ \wedge\ \widetilde{p}.type(\widetilde{\mathrm{MOF}}) : \textsc{Class} \ \wedge \\[1ex]
\qquad\quad class(\widetilde{o_1}) \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), ClassSort(\widetilde{p}.class(\widetilde{\mathrm{MOF}}))} \ \wedge\ \widetilde{o_1}.(\widetilde{p}.name) \neq \emptyset \ \wedge \\[1ex]
\qquad\quad \exists OID(OID \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), Oid\#MOF} \ \wedge\ OID \in \widetilde{o_1}.(\widetilde{p}.name) \ \wedge \\[1ex]
\qquad\qquad \forall \widetilde{o_2}(\widetilde{o_2} \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), Object\#MOF} \ \wedge\ \widetilde{o_2} \in \widetilde{\mathcal{M}} \ \rightarrow\ oid(\widetilde{o_2}) \neq OID) \\
\qquad\quad) \\
\qquad) \\
\quad) \\
\ ) \\
and \\
noBrokenLinks(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = true \quad otherwise;
\end{array}$$

and the *wellTypedLinks* is a function

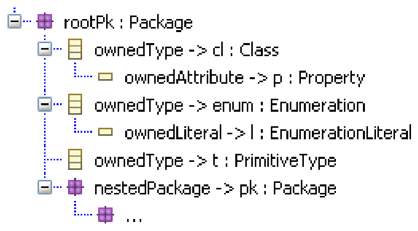

rootPk : Package
ownedType -> cl : Class
ownedAttribute -> p : Property
ownedType -> enum : Enumeration
ownedLiteral -> l : EnumerationLiteral
ownedType -> t : PrimitiveType
nestedPackage -> pk : Package
...

Figure 6.5: Tree structure of a metamodel $\widetilde{\mathcal{M}}$: MOF.

$$\mathit{wellTypedLinks} : [\![\mathrm{MOF}_0]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow [\![\textsc{Boolean}]\!]_{\mathrm{MOF}},$$

which checks that the nodes of a graph $\mathit{graph}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$, which are objects in $\widetilde{\mathcal{M}}$, are typed with the object types that participate in the corresponding property definition in $\widetilde{\mathrm{MOF}}$. This function is defined as follows:

$$\begin{array}{l}
\mathit{wellTypedLinks}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{false} \\
\quad \mathit{when}\ \exists(\widetilde{o_1}, \widetilde{o_2})\ (\widetilde{o_1}, \widetilde{o_2} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{Object\#MOF}}\ \wedge\ \widetilde{o_1}, \widetilde{o_2} \in \widetilde{\mathcal{M}}\ \wedge \\
\qquad \exists\widetilde{p}\ (\widetilde{p} : \textsc{Property}\ \wedge\ \widetilde{p} \in \widetilde{\mathrm{MOF}}\ \wedge\ \widetilde{p}.\mathit{type}(\widetilde{\mathrm{MOF}}) : \textsc{Class}\ \wedge \\
\qquad\quad \mathit{class}(\widetilde{o_1}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{ClassSort}(\widetilde{p}.\mathit{class}(\widetilde{\mathrm{MOF}}))}\ \wedge\ \widetilde{o_2} \in \widetilde{o_1}.(\widetilde{p}.\mathit{name})(\widetilde{\mathcal{M}})\ \wedge \\
\qquad\quad \mathit{class}(\widetilde{o_2}) \notin T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{ClassSort}(\widetilde{p}.\mathit{type}(\widetilde{\mathrm{MOF}}))} \\
\qquad) \\
\quad) \\
\mathit{and} \\
\mathit{wellTypedLinks}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true} \quad \mathit{otherwise}.
\end{array}$$

Definition 2 (MOF subgraph)**.** *Given two MOF graphs $\widetilde{G}' = (V'_{\mathcal{M}}, E'_{\mathcal{M}})$ and $\widetilde{G} = (V_{\mathcal{M}}, E_{\mathcal{M}})$, a pair $(\widetilde{G}', \widetilde{G})$ is called a MOF subgraph iff $V'_{\mathcal{M}} \subseteq V_{\mathcal{M}}$ and $E'_{\mathcal{M}} \subseteq E_{\mathcal{M}}$. We also denote a MOF subgraph $(\widetilde{G}', \widetilde{G})$ as $\widetilde{G}' \subseteq \widetilde{G}$.*

Tree structure

A MOF graph $(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$ can also be viewed as a tree of objects by considering the *containment properties* that are defined between object types in $\widetilde{\mathrm{MOF}}$. For example, a metamodel definition $\widetilde{\mathcal{M}}$ can be viewed as a tree, as shown in Fig. 6.5, where the root element is a PACKAGE instance $\widetilde{\mathit{rootPk}}$. This PACKAGE instance may contain either other PACKAGE instances through the containment meta-property *nestedPackage*, or TYPE instances through the containment meta-property *ownedType*. A nested PACKAGE instance may contain other PACKAGE instances recursively in an acyclic way. A TYPE instance can be either a CLASS instance, an ENUMERATION instance or a PRIMITIVETYPE instance. A CLASS instance may contain PROPERTY instances through the containment meta-property *ownedAttribute*. An ENUMERATION instance may contain ENUMERATIONLITERAL instances by means of the containment meta-property *ownedLiteral*. Therefore, the part of the metamodel definition that is shown in Fig. 6.4 can also be viewed as a tree, as shown in Fig. 6.6.

An edge in the tree view of a metamodel definition $\widetilde{\mathcal{M}}$ is defined by means of a containment property. A PROPERTY instance $\widetilde{p}$ is defined as a *containment* property by means of the function

$$\mathit{containment} : [\![\textsc{Property}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow [\![\textsc{Boolean}]\!]_{\mathrm{MOF}},$$

which is defined as follows[8]

[8]We assume that *isComposite* properties are always defined with an opposite property, the *containment property*, i.e., given a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF,

$$\forall\widetilde{p}\ (\widetilde{p} : \textsc{Property}\ \wedge\ \widetilde{p} \in \widetilde{\mathcal{M}}\ \wedge\ \widetilde{p}.\mathit{isComposite} = \mathit{true}\ \wedge\ \widetilde{p}.\mathit{opposite}(\widetilde{\mathcal{M}}) \neq \emptyset).$$

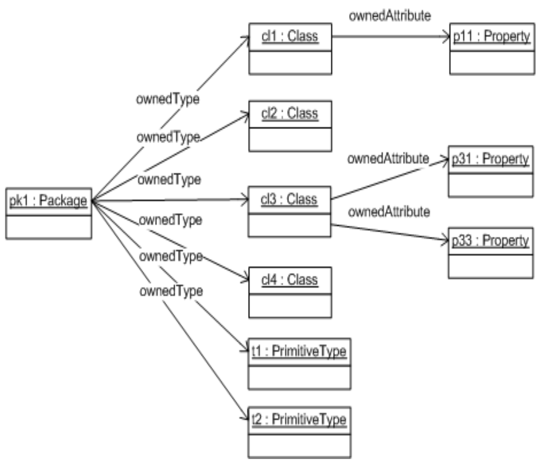


Figure 6.6: Tree view of part of the metamodel definition $\widetilde{\mathrm{MOF}}$.

$$containment(\widetilde{p}, \widetilde{\mathcal{M}}) = \begin{cases} true & when\ \widetilde{p} : \textsc{Property}\ \wedge\ \widetilde{\mathcal{M}} : \mathrm{MOF}\ \wedge\ \widetilde{p} \in \widetilde{\mathcal{M}}\ \wedge \\ & \quad (\widetilde{p}.opposite(\widetilde{\mathcal{M}}).isComposite = true) \\ false & otherwise \end{cases}$$

A relationship that is defined by means of a containment property between two objects, in a metamodel definition $\widetilde{\mathcal{M}}$, can be represented as a pair of the form $(\widetilde{o_1}, \widetilde{o_2})$, where $\widetilde{o_1}, \widetilde{o_2} \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), Object\#MOF}$, $\widetilde{o_1}, \widetilde{o_2} \in \widetilde{\mathcal{M}}$, $\widetilde{o_1}$ is the object that contains the object-typed property, and $\widetilde{o_2}$ is the object that is referred to by means of the property. The set of edges of this form defines a *containment relation* for the metamodel definition $\widetilde{\mathcal{M}}$. This set is defined for a specific collection $\widetilde{\mathcal{M}}$ of typed objects, such that $\widetilde{\mathcal{M}} : \mathrm{MOF}$, by means of the partial function

$$\begin{aligned} <_c :\ & [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \rightsquigarrow \\ & \mathcal{P}_{fin}(T_{reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), Object\#MOF} \times T_{reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), Object\#MOF}), \end{aligned}$$

where the $<_c$ function is only defined for pairs of the form $(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$, where $\widetilde{\mathcal{M}} : \mathrm{MOF}$ and $\widetilde{\mathrm{MOF}}$ is the definition of the MOF meta-metamodel, such that $\widetilde{\mathrm{MOF}} : \mathrm{MOF}$. The $<_c$ function is defined as follows:

$$\begin{aligned} & <_c(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \\ & \quad \{(\widetilde{o_1}, \widetilde{o_2}) \mid \widetilde{o_1}, \widetilde{o_2} : \textsc{Package}\ \wedge\ \widetilde{o_1}, \widetilde{o_2} \in \widetilde{\mathcal{M}}\ \wedge\ \widetilde{o_1} \neq \widetilde{o_2}\ \wedge \\ & \qquad \widetilde{o_1} \in \widetilde{o_2}.nestedPackage(\widetilde{\mathcal{M}})\}\ \cup \\ & \quad \{(\widetilde{o_1}, \widetilde{o_2}) \mid \widetilde{o_1} : \textsc{Type}\ \wedge\ \widetilde{o_2} : \textsc{Package}\ \wedge\ \widetilde{o_1}, \widetilde{o_2} \in \widetilde{\mathcal{M}}\ \wedge \\ & \qquad \widetilde{o_1} \in \widetilde{o_2}.ownedType(\widetilde{\mathcal{M}})\}\ \cup \\ & \quad \{(\widetilde{o_1}, \widetilde{o_2}) \mid \widetilde{o_1} : \textsc{Property}\ \wedge\ \widetilde{o_2} : \textsc{Class}\ \wedge\ \widetilde{o_1}, \widetilde{o_2} \in \widetilde{\mathcal{M}}\ \wedge \\ & \qquad \widetilde{o_1} \in \widetilde{o_2}.ownedAttribute(\widetilde{\mathcal{M}})\}\ \cup \\ & \quad \{(\widetilde{o_1}, \widetilde{o_2}) \mid \widetilde{o_1} : \textsc{EnumerationLiteral}\ \wedge\ \widetilde{o_2} : \textsc{Enumeration}\ \wedge\ \widetilde{o_1}, \widetilde{o_2} \in \widetilde{\mathcal{M}}\ \wedge \\ & \qquad \widetilde{o_1} \in \widetilde{o_2}.ownedLiteral(\widetilde{\mathcal{M}})\}. \end{aligned}$$

Note that this function is now defined for the metamodel definition $\widetilde{\mathrm{MOF}}$. Although the second argument is not used in the definition of the function, it is kept in order to generalize this function for any metamodel definition $\widetilde{\mathcal{M}}$ in the next section.

We define the associative, commutative function

$$\_ \cup \_ : [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \to [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$$

by means of the equation

$$\ll \mathit{ObjCol1} \gg \cup \ll \mathit{ObjCol2} \gg \,=\, \ll \mathit{ObjCol1}\ \mathit{ObjCol2} \gg,$$

where $\mathit{ObjCol1}, \mathit{ObjCol2} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{MOF}), \mathit{ObjectCollection}\{MOF\}}$, and the constructors $\_\ \_$ and $\ll \_ \gg$ denote the corresponding constructors of the MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ that were introduced earlier in this Section.

Definition 3 (MOF Tree)**.** *Given a collection* $\widetilde{\mathcal{M}}$ *of typed objects, such that* $\widetilde{\mathcal{M}} : \mathrm{MOF}$*, and the definition of the meta-metamodel* $\widetilde{\mathrm{MOF}}$*,*

$$(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}, <_c, \mathit{root}, \mathit{containments})$$

is called the MOF tree *of the metamodel definition* $\widetilde{\mathcal{M}}$*, also denoted by* $\mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$*, iff:*

(i) $\widetilde{\mathcal{M}}$ *is a set of typed objects that constitute the nodes in* $\mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$*.*

(ii) $\widetilde{\mathrm{MOF}}$ *is the definition of the MOF meta-metamodel.*

(iii) $<_c$ *is the partial function that defines the set of containment relationships in the metamodel definition* $\widetilde{\mathcal{M}}$ *as* $<_c (\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$*.*

(iv) $(\widetilde{\mathcal{M}}, \leqslant_c)$ *is a partially ordered set, where* $\leqslant_c$ *is a binary relation defined by the transitive-reflexive closure of the relation defined by the function* $<_c$*. That is, the relation* $\leqslant_c$ *is defined by the equation*

$$\leqslant_c \,=\, (<_c (\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}))^*.$$

Given two objects $\widetilde{o_1}$ *and* $\widetilde{o_2}$ *such that* $\widetilde{o_1}, \widetilde{o_2} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{Object\#MOF}}$ *and* $\widetilde{o_1}, \widetilde{o_2} \in \widetilde{\mathcal{M}}$*, we obtain the equivalences*

$$(\widetilde{o_1} <_c \widetilde{o_2}) \Longleftrightarrow (\widetilde{o_1}, \widetilde{o_2}) \in \,<_c (\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$$

and

$$(\widetilde{o_1} <_c^+ \widetilde{o_2}) \Longleftrightarrow (\widetilde{o_1}, \widetilde{o_2}) \in (<_c (\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}))^+,$$

where $(<_c (\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}))^+$ *is the transitive closure of the relation that is defined by* $<_c (\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$*.*

(v) The root *function obtains the root element of a MOF tree, and is defined as follows:*

$$\mathit{root} : [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \rightsquigarrow T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{Object\#MOF}}$$

$$\begin{aligned}\mathit{root}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = (\widetilde{o} \mid\ & \widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{Object\#MOF}} \ \wedge\ \widetilde{\mathcal{M}} : \mathrm{MOF} \ \wedge\ \widetilde{o} \in \widetilde{\mathcal{M}} \ \wedge \\ & \nexists \widetilde{o}' (\widetilde{o}' \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{Object\#MOF}} \ \wedge\ \widetilde{o}' \in \widetilde{\mathcal{M}} \ \wedge\ \widetilde{o} <_c^+ \widetilde{o}')).\end{aligned}$$

(vi) The function

$$\begin{aligned}&\mathit{containments} : T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{Object\#MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \rightsquigarrow \\ &\quad [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}\end{aligned}$$

obtains the children nodes of a specific node in a MOF tree. This function is only defined for tuples $(\widetilde{o}, \widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$*, where* $\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{Object\#MOF}}$*,* $\widetilde{\mathcal{M}} : \mathrm{MOF}$*, and* $\widetilde{o} \in \widetilde{\mathcal{M}}$ *as follows:*

$$\mathit{containments}(\widetilde{o}, \widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \bigcup\nolimits_{\widetilde{o}' \in \{\widetilde{o}' \mid \widetilde{o}' \in \widetilde{\mathcal{M}} \wedge \widetilde{o}' <_c \widetilde{o}\}} (\ll \widetilde{o}' \gg).$$

where $\widetilde{o}' \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{Object\#MOF}}$, and the union operator of collections of objects is defined above.

The data type $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ is defined by the set of collections of typed objects that are MOF trees. To denote that a collection of typed objects $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}} : \mathrm{MOF}$, is a MOF tree, we can also use the notation $\widetilde{\mathcal{M}} : \mathrm{MOF}$ by taking into account the following equivalence

$$\begin{aligned}\widetilde{\mathcal{M}} : \mathrm{MOF} \iff\quad & \mathit{graph}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) \textit{ is a } \mathrm{MOF} \textit{ graph} \ \wedge \\ & \mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) \textit{ is a } \mathrm{MOF} \textit{ tree}\end{aligned}$$

The $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ type is defined by means of a conditional membership axiom that refines the membership that preserves the graph structure of a collection of typed objects as follows:

$$\left.\begin{array}{l} \widetilde{\mathcal{M}} : \mathrm{MOF}_0 \;\wedge \\ \\ \mathit{noBrokenLinks}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true} \;\wedge \\ \\ \mathit{wellTypedLinks}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true} \;\wedge \\ \\ \mathit{singleContainer}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true} \;\wedge \\ \\ \mathit{singleRoot}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true} \end{array}\right\} \Longrightarrow \widetilde{\mathcal{M}} : \mathrm{MOF}$$

where *singleContainer* is a function

$$\mathit{singleContainer} : [\![\mathrm{MOF}_0]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow [\![\textsc{Boolean}]\!]_{\mathrm{MOF}},$$

which checks that an object in a tree $\mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$ cannot be contained in two different objects. This function is defined as follows:

$$\begin{array}{l}
\mathit{singleContainer}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{false} \\
\quad \mathit{when}\ \exists(\widetilde{o_1}, \widetilde{o_2})\ (\widetilde{o_1}, \widetilde{o_2} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\, \mathit{Object\#MOF}} \;\wedge\; \widetilde{o_1}, \widetilde{o_2} \in \widetilde{\mathcal{M}} \;\wedge\; \widetilde{o_1} \neq \widetilde{o_2} \;\wedge \\
\qquad \exists \widetilde{o_3}\ (\widetilde{o_3} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\, \mathit{Object\#MOF}} \;\wedge \\
\qquad\quad \widetilde{o_3} \in \widetilde{\mathcal{M}} \;\wedge\; \widetilde{o_3} \neq \widetilde{o_1} \;\wedge\; \widetilde{o_3} \neq \widetilde{o_2} \;\wedge \\
\qquad\quad \widetilde{o_3} <_c^+ \widetilde{o_1} \;\wedge\; \widetilde{o_3} <_c^+ \widetilde{o_2}) \\
\quad) \\
\mathit{and} \\
\mathit{singleContainer}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true} \quad \mathit{otherwise};
\end{array}$$

and *singleRoot* is a function

$$\mathit{singleRoot} : [\![\mathrm{MOF}_0]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow [\![\textsc{Boolean}]\!]_{\mathrm{MOF}},$$

which checks if a metamodel definition has a single root. The *singleRoot* function is defined as follows:

$$\begin{array}{l}
\mathit{singleRoot}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{false} \\
\quad \mathit{when}\ \exists \widetilde{o_1}\ (\widetilde{o_1} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\, \mathit{Object\#MOF}} \;\wedge\; \widetilde{\mathcal{M}} : \mathrm{MOF} \;\wedge\; \widetilde{o_1} \in \widetilde{\mathcal{M}} \;\wedge \\
\qquad \nexists \widetilde{o_2} (\widetilde{o_2} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\, \mathit{Object\#MOF}} \;\wedge\; \widetilde{o_2} \in \widetilde{\mathcal{M}} \;\wedge\; \widetilde{o_1} <_c^+ \widetilde{o_2}) \;\wedge \\
\qquad \exists \widetilde{o_3}\ (\widetilde{o_3} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\, \mathit{Object\#MOF}} \;\wedge\; \widetilde{\mathcal{M}} : \mathrm{MOF} \;\wedge\; \widetilde{o_3} \in \widetilde{\mathcal{M}} \;\wedge\; \widetilde{o_3} \neq \widetilde{o_1} \;\wedge \\
\qquad\quad \nexists \widetilde{o_4} (\widetilde{o_4} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\, \mathit{Object\#MOF}} \;\wedge\; \widetilde{o_4} \in \widetilde{\mathcal{M}} \;\wedge\; \widetilde{o_3} <_c^+ \widetilde{o_4}) \\
\quad) \\
\\
\mathit{and} \\
\mathit{singleRoot}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true} \quad \mathit{otherwise}.
\end{array}$$

Definition 4 (MOF subtree)**.** *Given two MOF trees $\widetilde{\mathcal{T}}'$ and $\widetilde{\mathcal{T}}$, such that $\widetilde{\mathcal{T}}', \widetilde{\mathcal{T}}$:* MOF*, a pair $(\widetilde{\mathcal{T}}', \widetilde{\mathcal{T}})$ is called a MOF subtree iff $(\widetilde{\mathcal{T}}', \widetilde{\mathcal{T}})$ is a MOF subgraph, i.e., $\widetilde{\mathcal{T}}' \subseteq \widetilde{\mathcal{T}}$.*

Note that we call *metamodel definition* $\widetilde{\mathcal{M}}$ to the values of the type $[\![\text{MOF}]\!]_{\text{MOF}}$. However, this values are collections of typed objects that can be viewed as graphs or trees, as shown above. OCL constraints are still not taken into account. Therefore, any collection of typed objects is a valid value for the type $[\![\text{MOF}]\!]_{\text{MOF}}$ if it preserves a graph structure and a tree structure. In Section 7, we define proper metamodel definitions as values of the type $[\![(\text{MOF}, \mathcal{C}_{MOF})]\!]_{\text{MOF}}$.

Type Definitions in $\widetilde{\textbf{MOF}}$

A model type permits defining collections of typed objects, i.e., model definitions. $[\![\text{MOF}]\!]_{\text{MOF}}$ can be considered as a model type whose values are model definitions of the form $\widetilde{\mathcal{M}} : \text{MOF}$. $[\![\text{MOF}]\!]_{\text{MOF}}$ is used as domain in the $\mathit{reflect}_{\text{MOF}}$ function, which permits formalizing a MOF metamodel in MEL automatically.

A metamodel definition $\widetilde{\mathcal{M}} : \text{MOF}$ can then be viewed as a collection that can contain nested subcollections recursively, where each subcollection is a subtree in $\mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}})$. Therefore, a subcollection of objects is constituted by the root node of the subtree and all the objects that are contained recursively by means of containment properties. Taking into account the containment relation $<_c$ that is defined by $<_c (\widetilde{\mathcal{M}}, \widetilde{\text{MOF}})$, a metamodel definition $\widetilde{\mathcal{M}} : \text{MOF}$ can be decomposed in different subtrees. These subtrees provide the information that is needed to define the types by means of the $\mathit{reflect}_{\text{MOF}}$ function. Given a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}} : \text{MOF}$, the types that may be algebraically represented in the $\mathit{reflect}_{\text{MOF}}(\widetilde{\mathcal{M}})$ theory are defined in the metamodel definition $\widetilde{\mathcal{M}}$ as follows:

Definition 5 (Object Type Definition)**.** $\widetilde{\text{OT}}$*, such that* $\widetilde{\text{OT}} : \text{MOF}$*, is called an object type definition in the metamodel definition* $\widetilde{\mathcal{M}}$ *iff satisfies the following conditions*

$$\mathit{root}(\widetilde{\text{OT}}, \widetilde{\text{MOF}}) : \textsc{Class} \ \wedge \ \mathit{tree}(\widetilde{\text{OT}}, \widetilde{\text{MOF}}) \subseteq \mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}}).$$

Definition 6 (Enumeration Type Definition)**.** $\widetilde{\text{ET}}$*, such that* $\widetilde{\text{ET}} : \text{MOF}$*, is called an enumeration type definition in the metamodel definition* $\widetilde{\mathcal{M}}$ *iff satisfies the following conditions*

$$\mathit{root}(\widetilde{\text{ET}}, \widetilde{\text{MOF}}) : \textsc{Enumeration} \ \wedge \ \mathit{tree}(\widetilde{\text{ET}}, \widetilde{\text{MOF}}) \subseteq \mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}}).$$

Definition 7 (Primitive Type Definition)**.** $\widetilde{\text{PT}}$*, such that* $\widetilde{\text{PT}} : \text{MOF}$*, is called a primitive type definition in the metamodel definition* $\widetilde{\mathcal{M}}$ *iff satisfies the following conditions*

$$\mathit{root}(\widetilde{\text{PT}}, \widetilde{\text{MOF}}) : \textsc{PrimitiveType} \ \wedge \ \mathit{tree}(\widetilde{\text{PT}}, \widetilde{\text{MOF}}) \subseteq \mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}}).$$

Definition 8 (Package Definition)**.** $\widetilde{\text{P}\textsc{k}}$*, such that* $\widetilde{\text{P}\textsc{k}} : \text{MOF}$*, is called a package definition in the metamodel definition* $\widetilde{\mathcal{M}}$ *iff satisfies the following conditions*

$$\mathit{root}(\widetilde{\text{P}\textsc{k}}, \widetilde{\text{MOF}}) : \textsc{Package} \ \wedge \ \mathit{tree}(\widetilde{\text{P}\textsc{k}}, \widetilde{\text{MOF}}) \subseteq \mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}}).$$

Definition 9 (Model Type Definition)**.** $\widetilde{\text{MT}}$*, such that* $\widetilde{\text{MT}} : \text{MOF}$*, is called a model type definition in the metamodel definition* $\widetilde{\mathcal{M}}$ *iff satisfies the following conditions*

$$\mathit{root}(\widetilde{\text{MT}}, \widetilde{\text{MOF}}) : \textsc{Package} \ \wedge \ \mathit{tree}(\widetilde{\text{MT}}, \widetilde{\text{MOF}}) = \mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}}).$$

The metamodel definition $\widetilde{\text{MOF}} : \text{MOF}$ can also be decomposed as a collection of object type definitions, where the set of object type definitions is { $\widetilde{\textsc{NamedElement}}$, $\widetilde{\textsc{Package}}$, $\widetilde{\textsc{Type}}$, $\widetilde{\textsc{Class}}$, $\widetilde{\textsc{Property}}$, $\widetilde{\textsc{Enumeration}}$, $\widetilde{\textsc{EnumerationLiteral}}$, $\widetilde{\textsc{PrimitiveType}}$ }, and the subcollection of primitive type definitions is { $\widetilde{\textsc{Boolean}}$, $\widetilde{\textsc{String}}$, $\widetilde{\textsc{Integer}}$, $\widetilde{\textsc{Real}}$ }.

The Specialization Relation $<_s$

A *specialization* relationship is a taxonomic relationship between two object types. This relationship permits specializing a general object type into a more specific one. A specialization relationship between an object type $\widetilde{\text{OT}}_1 : \text{MOF}$ and an object type $\widetilde{\text{OT}}_2 : \text{MOF}$ is denoted by $\widetilde{\text{OT}}_1 <_s \widetilde{\text{OT}}_2$, where $\widetilde{\text{OT}}_1$ is the specialized object type definition, and $\widetilde{\text{OT}}_2$ is the supertype definition.

The *specialization* relation $<_s$ is defined for pairs of object types as a subset inclusion between the carriers of the corresopnding sorts in the initial algebra of the $\mathit{reflect}_{\text{MOF}}(\widetilde{\text{MOF}})$ theory as follows:

$$\begin{array}{l}
[\![\textsc{Type} <_s \textsc{NamedElement}]\!]_{\mathrm{MOF}} \Longrightarrow \\
\quad [\![\textsc{Type}]\!]_{\mathrm{MOF}} \subseteq [\![\textsc{NamedElement}]\!]_{\mathrm{MOF}} \\[1em]
[\![\textsc{Package} <_s \textsc{NamedElement}]\!]_{\mathrm{MOF}} \Longrightarrow \\
\quad [\![\textsc{Package}]\!]_{\mathrm{MOF}} \subseteq [\![\textsc{NamedElement}]\!]_{\mathrm{MOF}} \\[1em]
[\![\textsc{Property} <_s \textsc{NamedElement}]\!]_{\mathrm{MOF}} \Longrightarrow \\
\quad [\![\textsc{Property}]\!]_{\mathrm{MOF}} \subseteq [\![\textsc{NamedElement}]\!]_{\mathrm{MOF}} \\[1em]
[\![\textsc{EnumerationLiteral} <_s \textsc{NamedElement}]\!]_{\mathrm{MOF}} \Longrightarrow \\
\quad [\![\textsc{EnumerationLiteral}]\!]_{\mathrm{MOF}} \subseteq [\![\textsc{NamedElement}]\!]_{\mathrm{MOF}} \\[1em]
[\![\textsc{Class} <_s \textsc{Type}]\!]_{\mathrm{MOF}} \Longrightarrow \\
\quad [\![\textsc{Class}]\!]_{\mathrm{MOF}} \subseteq [\![\textsc{Type}]\!]_{\mathrm{MOF}} \\[1em]
[\![\textsc{DataType} <_s \textsc{Type}]\!]_{\mathrm{MOF}} \Longrightarrow \\
\quad [\![\textsc{DataType}]\!]_{\mathrm{MOF}} \subseteq [\![\textsc{Type}]\!]_{\mathrm{MOF}} \\[1em]
[\![\textsc{Enumeration} <_s \textsc{DataType}]\!]_{\mathrm{MOF}} \Longrightarrow \\
\quad [\![\textsc{Enumeration}]\!]_{\mathrm{MOF}} \subseteq [\![\textsc{DataType}]\!]_{\mathrm{MOF}} \\[1em]
[\![\textsc{PrimitiveType} <_s \textsc{DataType}]\!]_{\mathrm{MOF}} \Longrightarrow \\
\quad [\![\textsc{PrimitiveType}]\!]_{\mathrm{MOF}} \subseteq [\![\textsc{DataType}]\!]_{\mathrm{MOF}}
\end{array}$$

An object $\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),Object\#MOF}$ in a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF, may be an instance of different object types. In addition, we can state that any object that may participate in a metamodel definition $\widetilde{\mathcal{M}}$: MOF is an instance of the NamedElement object type, i.e.,

$$\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),Object\#MOF} \Leftrightarrow \widetilde{o} : \textsc{NamedElement},$$

since NamedElement is the common supertype of all the other object types that are defined in $\widetilde{\mathrm{MOF}}$.

Additional Semantics

In the current definition of the $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ type, objects can be defined in a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF, as instances of object types. For instance, the term

```
< oid#Class('Class0) : Class |
    name : "Type", isAbstract : false,
    package : oid#Package('Package0),
    superClass : OrderedSet{ oid#Class('NamedElement) },
    ownedAttribute : empty-orderedset#MOF
>
```

represents the Class instance $\widetilde{cl}$, such that $\widetilde{cl}$: Class and $\widetilde{cl} = root(\widetilde{\textsc{Type}}, \widetilde{\mathrm{MOF}})$, where $\widetilde{\textsc{Type}}$, such that $\widetilde{\textsc{Type}}$: MOF, is the definition of the Type object type. However, it is also feasible to define property values in this object using properties that do not belong to the Class object type. For example,

```
< oid#Class('Type) : Class |
    name : "Type", isAbstract : false,
    package : oid#Package('Package0),
    superClass : OrderedSet{ oid#Class('NamedElement) },
    ownedAttribute : empty-orderedset#MOF,
    type
>.
```

In a specific metamodel definition $\widetilde{\mathcal{M}}$, objects can be defined either with *unset* properties, those that are not initialized, or with *set* properties, those that are initialized with a suitable value. To distinguish if a property is *unset* or *set* in a specific object definition, this has to be defined in the object. In the current definition of the $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ type, defining an object without indicating all the properties that are defined for the corresponding object type is allowed. For example, `< oid#Class('Type) : Class | noneProperty#MOF >` is a valid CLASS instance. Therefore, we cannot know whether the meta-property `name` is initialized or not in this object.

These two problems are addressed by means of constraints that are added to the membership that is defined in the `MOFSTRUCTURE` theory. We introduce some auxiliar domains and functions that are used to define these constraints. We define the sets of types D' and D as:

$$\begin{array}{ll} \mathrm{D}' = & \{ \texttt{Bool, String, Int, Float, Oid, Object\#MOF} \} \\ \mathrm{D} = & \bigcup_{\texttt{T} \in \mathrm{D}'} \{ \texttt{NeSet\{T\}, Set\{T\}, NeOrderedSet\{T\}, OrderedSet\{T\},} \\ & \texttt{NeBag\{T\}, Bag\{T\}, NeSequence\{T\}, Sequence\{T\},} \\ & \texttt{Collection\{T\}} \} \cup \mathrm{D}' \end{array}$$

where $\texttt{T} \in \mathrm{D}'$. The set of values that can be used to define object properties in a specific object is defined as

$$\mathcal{D} = \bigcup_{t \in \mathrm{D}} (T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),t}).$$

The membership of the `MOFSTRUCTURE` theory that has been presented above is refined with two more constraints, which are defined by means of the following functions:

- *validProperties:* The function

$$\mathit{validProperties} : [\![\mathrm{MOF}_0]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow [\![\mathrm{BOOLEAN}]\!]_{\mathrm{MOF}},$$

  checks if every object in a metamodel definition $\widetilde{\mathcal{M}}$ has only properties that are defined in its object type or in any of its supertypes, taking the specialization relation $<_s$ into account. This function is defined as follows:

$$\begin{array}{l} \mathit{validProperties}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{false} \\ \\ \mathit{when}\ \exists(\widetilde{o})\ (\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\mathit{Object\#MOF}} \ \wedge\ \widetilde{o} \in \widetilde{\mathcal{M}} \ \wedge \\ \quad (\\ \qquad \exists(\mathit{prop} : \mathit{value})(\mathit{prop} \in \mathit{OpNames} \ \wedge\ \mathit{value} \in \mathcal{D} \ \wedge \\ \\ \qquad\quad (\mathit{prop} : \mathit{value}) \in \mathit{getProperties}(\widetilde{o}) \ \wedge\ \forall\widetilde{p}(\widetilde{p} : \mathrm{PROPERTY} \ \wedge\ \widetilde{p} \in \widetilde{\mathrm{MOF}} \ \wedge \\ \\ \qquad\qquad \mathit{class}(\widetilde{o}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\mathit{ClassSort}(\widetilde{p}.\mathit{class}(\widetilde{\mathrm{MOF}}))} \ \rightarrow\ \widetilde{p}.\mathit{name} \neq \mathit{prop} \\ \qquad\quad) \\ \qquad) \\ \quad \vee \\ \qquad \exists \mathit{prop}(\mathit{prop} \in \mathit{OpNames} \ \wedge\ \mathit{prop} \in \mathit{getProperties}(\widetilde{o}) \ \wedge \\ \\ \qquad\quad \forall\widetilde{p}(\widetilde{p} : \mathrm{PROPERTY} \ \wedge\ \widetilde{p} \in \widetilde{\mathrm{MOF}} \ \wedge \\ \\ \qquad\qquad \mathit{class}(\widetilde{o}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\mathit{ClassSort}(\widetilde{p}.\mathit{class}(\widetilde{\mathrm{MOF}}))} \ \rightarrow\ \widetilde{p}.\mathit{name} \neq \mathit{prop} \\ \qquad\quad) \\ \qquad) \\ \quad) \\ \mathit{and} \\ \mathit{validProperties}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true} \quad \mathit{otherwise}. \end{array}$$

- *allProperties:* For every object $\widetilde{o}$ in a metamodel definition $\widetilde{\mathcal{M}}$, the function

$$\mathit{allProperties} : [\![\mathrm{MOF}_0]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow [\![\mathrm{BOOLEAN}]\!]_{\mathrm{MOF}},$$

  checks if $\widetilde{o}$ contains all the properties that are defined either for its object type in $\widetilde{\mathrm{MOF}}$ or for any of its supertypes. This function is defined as follows:

$$
\begin{array}{l}
\mathit{allProperties}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{false} \\
\\
\mathit{when}\ \exists(\widetilde{o})\ (\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{Object\#MOF}}\ \wedge\ \widetilde{o} \in \widetilde{\mathcal{M}}\ \wedge \\
\\
\quad \exists \widetilde{p}(\widetilde{p} : \textsc{Property}\ \wedge\ \widetilde{p} \in \widetilde{\mathrm{MOF}}\ \wedge\ \mathit{class}(\widetilde{o}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{ClassSort}(\widetilde{p}.\mathit{class}(\widetilde{\mathrm{MOF}}))}\ \wedge \\
\quad\quad (\\
\quad\quad\quad \nexists(\mathit{prop} : \mathit{value})(\mathit{prop} \in \mathit{OpNames}\ \wedge\ \mathit{value} \in \mathcal{D}\ \wedge \\
\\
\quad\quad\quad\quad \widetilde{p}.\mathit{name} = \mathit{prop}\ \wedge\ (\mathit{prop} : \mathit{value}) \in \mathit{getProperties}(\widetilde{o}) \\
\quad\quad\quad) \\
\quad\quad \wedge \\
\quad\quad\quad \nexists(\mathit{prop})(\mathit{prop} \in \mathit{OpNames}\ \wedge\ \widetilde{p}.\mathit{name} = \mathit{prop}\ \wedge\ (\mathit{prop}) \in \mathit{getProperties}(\widetilde{o})) \\
\quad\quad) \\
\quad) \\
\mathit{and} \\
\mathit{allProperties}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true} \quad \mathit{otherwise}.
\end{array}
$$

The membership axiom that defines the $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ data type in the `MOFSTRUCTURE` theory is as follows:

$$
\boxed{\left.\begin{array}{l}
\widetilde{\mathcal{M}} : \mathrm{MOF}_0\ \wedge\ \widetilde{\mathrm{MOF}} : \mathrm{MOF}\ \wedge \\
\\
\mathit{noBrokenLinks}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true}\ \wedge \\
\\
\mathit{wellTypedLinks}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true}\ \wedge \\
\\
\mathit{singleContainer}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true}\ \wedge \\
\\
\mathit{singleRoot}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true}\ \wedge \\
\\
\mathit{validProperties}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true}\ \wedge \\
\\
\mathit{allProperties}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true}
\end{array}\right\} \Longrightarrow \widetilde{\mathcal{M}} : \mathrm{MOF}}
$$

In this section, we have presented the result of the function $\mathit{reflect}_{\mathrm{MOF}}$ for a *single* input value: the metamodel definition $\widetilde{\mathrm{MOF}}$. In the following section, the $\mathit{reflect}_{\mathrm{MOF}}$ function is defined for any metamodel definition $\widetilde{\mathcal{M}}$ to provide the algebraic semantics for the different kinds of types that can be defined in a metamodel definition $\widetilde{\mathcal{M}} : \mathrm{MOF}$, i.e., for model type definitions $\widetilde{\mathcal{M}}$, for package definitions $\widetilde{\mathrm{P{\small K}}}$, for primitive type definitions $\widetilde{\mathrm{PT}}$, for enumeration type definitions $\widetilde{\mathrm{ET}}$, and for object type definitions $\widetilde{\mathrm{OT}}$. In addition, the $\mathit{reflect}_{\mathrm{MOF}}$ function also defines the algebraic semantics of both the specialization relation $<_s$ and the containment relation $<_c$ for each metamodel definition $\widetilde{\mathcal{M}}$. Therefore, the $\mathit{reflect}_{\mathrm{MOF}}$ function generalizes, in a precise, mechanical way, the formalization process of the metamodel definition given for $\widetilde{\mathrm{MOF}}$ to any metamodel definition $\widetilde{\mathcal{M}} : \mathrm{MOF}$.

6.2.4 Graphical Representation of MOF Metamodel Definitions $\widetilde{\mathcal{M}}$

Both the metamodel definition $\widetilde{\mathrm{MOF}}$ and the metamodel definition $\widetilde{\mathrm{UML}}$ reuse a basic infrastructure of object type definitions [22]. The UML standard specification [89] provides a graphical concrete syntax for packages, primitive type definitions, enumeration type definitions and object type definitions, in the form class diagrams.

As an example, the class diagram in Fig. 6.7 represents the metamodel definition $\widetilde{\mathrm{RDBMS}}$. In the figure, the graphical elements are classified by their object types to provide an intuition of how object type instances of a metamodel definition $\widetilde{\mathcal{M}}$ are represented in a class diagram. A detailed explanation of the mapping of the graphical concrete syntax into the abstract syntax of UML and, thereby, into the abstract syntax of MOF is provided in [89].

However, model types do not have a standard graphical representation in a class diagram, because this concept has not been defined. In this paper, we graphically represent a model type definition in two ways:

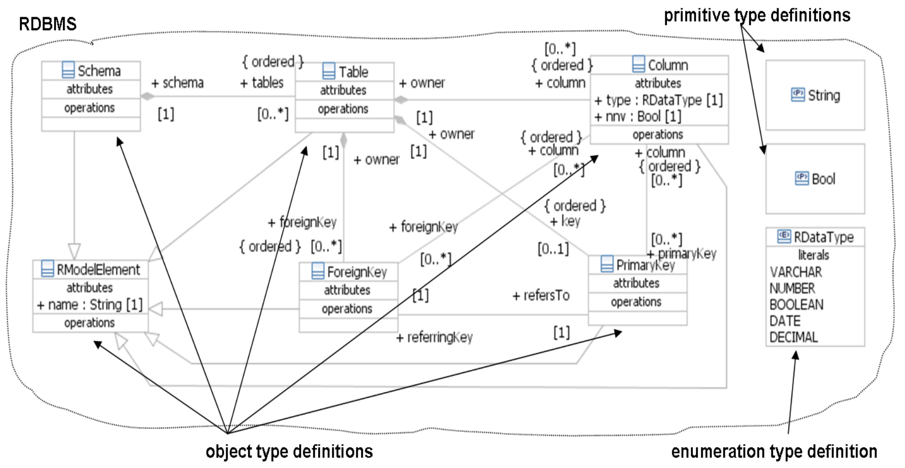


Figure 6.7: Graphical representation of the model type definition $\widetilde{\text{RDBMS}}$.

- as a boundary, in the form of a dotted line, for the object type instances that appear in a class diagram; or
- as a boundary, in the form of a dotted line, for the PACKAGE instances in a package diagram.

In both cases, the boundary line is tagged with a name representing the name of the model type definition. In this way, we can present a metamodel definition $\widetilde{\mathcal{M}}$ either as a term such that $\widetilde{\mathcal{M}}$: MOF, or as a MOF class diagram, taking into account that both are isomorphic syntactical representations.

6.3 Algebraic Semantics of MOF Metamodels Static Structure

SpecMEL is the data type of *finitely-presented* MEL theories, that is, theories of the form $(S, <, \Omega, E \cup A)$, where all the components are finite. Without loss of generality we assume countable sets *Sorts*, *OpNames*, *VarNames*, and *Ops*, so that:

- each set of sorts S is a finite subset of *Sorts*;
- the operator names in Ω are a finite subset of *OpNames*;
- all variables appearing in $E \cup A$ belong to the set *Vars*, where

$$Vars = \{x : s \mid x \in VarNames, s \in Sorts\} \ \cup \ \{x : [s] \mid x \in VarNames, s \in Sorts\};$$

- and *Ops* is the set of operators that can be defined in SpecMEL, which is defined by:

$$Ops = \{(f : s_1 \times \cdots \times s_n \rightarrow s) \mid f \in OpNames \ \ \wedge \ \ s, s_1, \ldots, s_n \in Sorts\}.$$

As shown in the previous section, the $\mathit{reflect}_{\text{MOF}}(\widetilde{MOF})$ theory provides the algebraic representation of the types that are defined in the MOF meta-metamodel $\widetilde{\text{MOF}}$. In particular, this theory provides the $[\![\text{MOF}]\!]_{\text{MOF}}$ data type, whose values are collections of typed objects that can be viewed as either MOF graphs or MOF trees. This type is used as domain of the function

$$\mathit{reflect}_{\text{MOF}} : [\![\text{MOF}]\!]_{\text{MOF}} \longrightarrow \mathit{SpecMEL}$$

so that a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF, can be mapped to a MEL theory $(S, <, \Omega, E \cup A)$. In particular, note that $\widetilde{\text{MOF}}$: MOF, so that the theory $\mathit{reflect}_{\text{MOF}}(\widetilde{\text{MOF}})$ is a specific application of the $\mathit{reflect}_{\text{MOF}}$ function. However, to break the self-reference in the definition of the $\mathit{reflect}_{\text{MOF}}$ function, we treat the metamodel definition $\widetilde{\text{MOF}}$ as a special case.

We introduce the notation $[\![\mathrm{OT}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ to define a subset of the cartesian product $[\![\mathrm{OT}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$, where OT is the sort of a specific object type, such as NamedElement for example. We define the domain $[\![\mathrm{OT}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ as a subset

$$[\![\mathrm{OT}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \subseteq [\![\mathrm{OT}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}},$$

where $(\widetilde{o}, \widetilde{\mathcal{M}}) \in [\![\mathrm{OT}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ iff $\widetilde{o} : \mathrm{OT}$, $\widetilde{\mathcal{M}} : \mathrm{MOF}$ and $\widetilde{o} \in \widetilde{\mathcal{M}}$. Given the metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}} : \mathrm{MOF}$. A $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory is defined by instantiating the `EXT-MODEL{OBJ :: TH-OBJECT}` theory, shown in Fig. 6.1, with a theory that is defined for a metamodel definition $\widetilde{\mathcal{M}}$ by means of the function

$$\mathit{defineParameter} : [\![\textsc{NamedElement}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow \mathit{SpecMEL}.$$

More specifically, the $\mathit{defineParameter}(\mathit{root}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}), \widetilde{\mathcal{M}})$ theory acts as actual parameter for the `EXT-MODEL{OBJ :: TH-OBJECT}` theory, where $\widetilde{\mathcal{M}} : \mathrm{MOF}$ and $\mathit{root}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) : \textsc{Package}$.

To instantiate the `EXT-MODEL{OBJ :: TH-OBJECT}` theory, we define a signature morphism, called $\mathcal{M}$, that maps the sorts and operators of the formal parameter theory `TH-OBJECT` to the sorts and operators of the actual parameter theory $\mathit{defineParameter}(\mathit{root}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}), \widetilde{\mathcal{M}})$ for a specific metamodel definition $\widetilde{\mathcal{M}}$. This signature morphism is called *view* in Maude, and is defined, in Maude notation, for a specific metamodel definition $\widetilde{\mathcal{M}}$ as follows:

```
view [M] from TH-EOBJECT to [mod#M] is
   sort Cid to [Cid#M] .
   sort Object to [Object#M] .
   sort ObjectOid to [Oid#M] .
   sort Property to [Property#M] .
   sort PropertySet to [PropertySet#M] .
   op noneProperty to [noneProperty#M] .
   op nullEObject to [nullObject#M] .
endv
```

where $\mathcal{M}$ is the name of the root package of the metamodel definition $\widetilde{\mathcal{M}}$. The $\mathcal{M}$ view permits instantiating the `EXT-MODEL{OBJ :: TH-OBJECT}` theory by means of the expression `EXT-MODEL{`$\mathcal{M}$`}`. Therefore, the $\mathit{reflect}_{\mathrm{MOF}}$ function is defined for a given metamodel definition $\widetilde{\mathcal{M}}$ as follows:

$$\mathit{reflect}_{\mathtt{MOF}}(\widetilde{\mathcal{M}}) = \mathtt{EXT\text{-}MODEL}\{\mathcal{M}\},$$

where $\widetilde{\mathcal{M}} \neq \widetilde{\mathrm{MOF}}$. When $\widetilde{\mathcal{M}} = \widetilde{\mathrm{MOF}}$, the $\mathit{reflect}_{\mathrm{MOF}}$ function is defined as

$$\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}) = \mathtt{MODEL\{MOF\}} \cup \mathtt{MOFSTRUCTURE},$$

where `MOF` is the view that is defined for the metamodel definition $\widetilde{\mathrm{MOF}}$, and `MOFSTRUCTURE` is the theory that provides the membership that defines both the graph and the tree structure of a metamodel definition $\widetilde{\mathcal{M}}$ such that $\widetilde{\mathcal{M}} : \mathrm{MOF}$, as shown in Section 6.2.

In subsequent sections, we provide: (a) the generic semantics for any metamodel definition $\widetilde{\mathcal{M}}$ in the `EXT-MODEL{OBJ :: TH-OBJECT}` theory, providing the semantics of model types; and (b) the specific semantics of a specific metamodel definition $\widetilde{\mathcal{M}}$ in the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory, by defining the *defineParameter* function for: packages, enumeration types and object types.

6.3.1 Generic Semantics of any Metamodel Definition $\widetilde{\mathcal{M}}$

In this section we generalize for any metamodel $\mathcal{M}$ the structure of graph and tree that has been defined for metamodel definitions $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}} : \mathrm{MOF}$, to model definitions $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$. The graph and tree structure that is defined for a model type $\mathcal{M}$ is provided by means of a conditional membership axiom of the form

$$\widetilde{M} : \mathcal{M}_0 \ \wedge \ \mathit{condition}_1 = \mathit{true} \ \wedge \cdots \wedge \ \mathit{condition}_n = \mathit{true} \implies \widetilde{M} : \mathcal{M},$$

which indicates that a collection $\widetilde{M}$ of typed objects belongs to the carrier of the $\mathcal{M}$ sort if $\widetilde{M}$ keeps both a graph structure, by considering the object-typed properties that are defined in $\widetilde{\mathcal{M}}$, and a tree structure, by considering the containment properties that are defined in $\widetilde{\mathcal{M}}$. The graph/tree structure is checked by means of the conditions $condition_1 \dots condition_n$ of the membership.

The MOF type can be considered as a model type, whose values are metamodel definitions $\widetilde{\mathcal{M}}$. Therefore, this membership is also applied for the metamodel definition $\widetilde{\text{MOF}}$

$$\widetilde{\mathcal{M}} : \text{MOF}_0 \ \wedge \ condition_1 = true \ \wedge \cdots \wedge \ condition_n = true \implies \widetilde{\mathcal{M}} : \text{MOF},$$

To define the conditions $condition_1 \dots condition_n$, we need the type MOF, which is defined by means of the previous membership. Again, we find a self-referential definition, which we break by following a two step strategy:

(i) the semantics of the MOF model type is provided by means of an adhoc membership in the $reflect_{\text{MOF}}(\widetilde{\text{MOF}})$ theory, which has already been presented in Section 6.2; and

(ii) the semantics of a model type $\mathcal{M}$, which is different from MOF, is provided by means of a generic membership in the `EXT-MODEL{OBJ :: TH-OBJECT}` theory. In the definition of this membership, the model type MOF is also used. In this section, we provide generic concepts to define: (i) the graph and (ii) the tree structure of a model definition, and (iii) the generic semantics of a model type $\mathcal{M}$.

Graph Structure

Given a specific metamodel definition $\widetilde{\mathcal{M}}$, an edge in the graph of a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, is represented as a pair $(\widetilde{o_1}, \widetilde{o_2})$, where $\widetilde{o_1}, \widetilde{o_2} \in T_{reflect_{\text{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}}$, $\widetilde{o_1}, \widetilde{o_2} \in \widetilde{M}$, $\widetilde{o_1}$ is the object that contains the object-typed property, and $\widetilde{o_2}$ is the object that is referred to by means of the property. Given a collection of typed objects $\widetilde{M}$, the collection of edges that are defined is given by the partial function

$$\begin{aligned} edges : \ & [\![\mathcal{M}]\!]_{\text{MOF}} \times [\![\text{MOF}]\!]_{\text{MOF}} \rightsquigarrow \\ & \mathcal{P}_{fin}(T_{reflect_{\text{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}} \times T_{reflect_{\text{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}}), \end{aligned}$$

which is only defined, for pairs of the form $(\widetilde{M}, \widetilde{\mathcal{M}})$, where: $\widetilde{M} : \mathcal{M}$, and $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF, is the corresponding metamodel definition. In this case, the *edges* function is defined by the mapping:

$$\begin{aligned} edges(\widetilde{M}, \widetilde{\mathcal{M}}) = \ \{ (\widetilde{o_1}, \widetilde{o_2}) \mid \ & \widetilde{o_1}, \widetilde{o_2} \in T_{reflect_{\text{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}} \ \wedge \ \widetilde{o_1}, \widetilde{o_2} \in \widetilde{M} \ \wedge \\ & \exists \widetilde{p} \ (\widetilde{p} : \text{Property} \ \wedge \ \widetilde{p} \in \widetilde{\mathcal{M}} \ \wedge \\ & \quad \widetilde{p}.type(\widetilde{\mathcal{M}}) : \text{Class} \ \wedge \\ & \quad \widetilde{o_2} \in \widetilde{o_1}.(\widetilde{p}.name)(\widetilde{M})) \\ \}. & \end{aligned}$$

Definition 10 (Model Graph)**.** *Given a collection $\widetilde{M}$ of typed objects, such that $\widetilde{M} : \mathcal{M}$, and the corresponding metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$:* MOF*, $graph(\widetilde{M}, \widetilde{\mathcal{M}})$ is called the* model graph *of the model definition $\widetilde{M}$ and is defined by the equation $graph(\widetilde{M}, \widetilde{\mathcal{M}}) = (V_M, E_M)$, where:*

(i) V_M is a collection of typed objects that constitutes the set of nodes of the graph $graph(\widetilde{M}, \widetilde{\mathcal{M}})$, and is defined by the equation $V_M = \widetilde{M}$; and

(ii) E_M is the set of edges of the graph $graph(\widetilde{M}, \widetilde{\mathcal{M}})$, and is defined by the equation

$$E_M = edges(\widetilde{M}, \widetilde{\mathcal{M}}).$$

Note that a MOF graph $graph(\widetilde{\mathcal{M}_{\text{MOF}}}, \widetilde{\text{MOF}})$, such that $\widetilde{\mathcal{M}_{\text{MOF}}}$: MOF and $\widetilde{\text{MOF}}$: MOF, is also a model graph $graph(\widetilde{M}, \widetilde{\mathcal{M}})$, such that $\widetilde{M} : \mathcal{M}$ and $\widetilde{\mathcal{M}}$: MOF, where $\widetilde{M} = \widetilde{\mathcal{M}_{\text{MOF}}}$ and $\widetilde{\mathcal{M}} = \widetilde{\text{MOF}}$.

Definition 11 (Model subgraph)**.** *Given a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$:* MOF*, and two model definitions $\widetilde{M}'$ and $\widetilde{M}$, such that $\widetilde{M}', \widetilde{M} : \mathcal{M}$, a pair $(graph(\widetilde{M}', \widetilde{\mathcal{M}}), graph(\widetilde{M}, \widetilde{\mathcal{M}}))$ is called a model subgraph iff $graph(\widetilde{M}', \widetilde{\mathcal{M}}) = (V'_M, E'_M)$, $graph(\widetilde{M}, \widetilde{\mathcal{M}}) =$*

(V_M, E_M), $V'_{\mathcal{M}} \subseteq V_{\mathcal{M}}$ and $E'_{\mathcal{M}} \subseteq E_{\mathcal{M}}$. We also denote a model subgraph $(graph(\widetilde{M}', \widetilde{\mathcal{M}}), graph(\widetilde{M}, \widetilde{\mathcal{M}}))$ by means of the expression

$$graph(\widetilde{M}', \widetilde{\mathcal{M}}) \subseteq graph(\widetilde{M}, \widetilde{\mathcal{M}}).$$

Tree structure

An edge in the tree view of a model definition $\widetilde{M}$ is defined by means of a containment property, which is defined in the corresponding metamodel definition $\widetilde{\mathcal{M}}$. A Property instance $\widetilde{p}$ is defined as a *containment* property by means of the function

$$containment : [\![\text{Property}]\!]_{\text{MOF}} \otimes [\![\text{MOF}]\!]_{\text{MOF}} \longrightarrow [\![\text{Boolean}]\!]_{\text{MOF}},$$

which is defined as follows[9]:

$$containment(\widetilde{p}, \widetilde{\mathcal{M}}) = \begin{cases} true & when\ \widetilde{p} : \text{Property} \ \wedge\ \widetilde{\mathcal{M}} : \text{MOF} \ \wedge\ \widetilde{p} \in \widetilde{\mathcal{M}} \ \wedge \\ & \quad (\widetilde{p}.opposite(\widetilde{\mathcal{M}}).isComposite = true) \\ false & otherwise \end{cases}$$

Recall the function $ClassSort : \text{Class} \rightarrow Sorts$, which obtains the sort that corresponds to a MOF Class instance as follows: $ClassSort : \widetilde{cl} \mapsto \widetilde{cl}.name$, where $\widetilde{cl} : \text{Class}$. A relationship that is defined by means of a containment property between two objects, in a model definition $\widetilde{M}$, can be represented as a pair of the form $(\widetilde{o_1}, \widetilde{o_2})$, where $\widetilde{o_1}, \widetilde{o_2} \in T_{reflect_{\text{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}}$, $\widetilde{o_1}, \widetilde{o_2} \in \widetilde{M}$, $\widetilde{o_1}$ is the object that contains the object-typed property, and $\widetilde{o_2}$ is the object that is referred to by means of the property. The set of edges of this form defines a *containment relation* $<_c$ for the model definition $\widetilde{M}$. This set is defined for a specific collection $\widetilde{M}$ of typed objects, such that $\widetilde{M} : \mathcal{M}$, by means of the partial function

$$<_c : [\![\mathcal{M}]\!]_{\text{MOF}} \times [\![\text{MOF}]\!]_{\text{MOF}} \rightsquigarrow$$

$$\mathcal{P}_{fin}(T_{reflect_{\text{MOF}}(\widetilde{\text{MOF}}), Object\#MOF} \times T_{reflect_{\text{MOF}}(\widetilde{\text{MOF}}), Object\#MOF}),$$

where the $<_c$ function is only defined for tuples of the form $(\widetilde{M}, \widetilde{\mathcal{M}})$, where $\widetilde{M} : \mathcal{M}$, and $\widetilde{\mathcal{M}}$ is the corresponding metamodel definition such that $\widetilde{\mathcal{M}} : \text{MOF}$. The $<_c$ function is defined as follows:

$$\begin{aligned} &<_c (\widetilde{M}, \widetilde{\mathcal{M}}) = \\ &\{(\widetilde{o_1}, \widetilde{o_2}) \mid \quad \widetilde{o_1}, \widetilde{o_2} \in T_{reflect_{\text{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}} \ \wedge\ \widetilde{M} : \mathcal{M} \ \wedge\ \widetilde{\mathcal{M}} : \text{MOF} \ \wedge\ \widetilde{o_1}, \widetilde{o_1} \in \widetilde{M} \ \wedge \\ &\qquad \exists \widetilde{p}\ (\widetilde{p} : \text{Property} \ \wedge\ \widetilde{p} \in \widetilde{\mathcal{M}} \ \wedge\ containment(\widetilde{p}, \widetilde{\mathcal{M}}) = true \ \wedge \\ &\qquad class(\widetilde{o_1}) \in T_{reflect_{\text{MOF}}(\widetilde{\mathcal{M}}), ClassSort(\widetilde{p}.class(\widetilde{\mathcal{M}}))} \ \wedge \\ &\qquad class(\widetilde{o_2}) \in T_{reflect_{\text{MOF}}(\widetilde{\mathcal{M}}), ClassSort(\widetilde{p}.type(\widetilde{\mathcal{M}}))} \ \wedge\ \widetilde{o_2} \in \widetilde{o_1}.(\widetilde{p}.name)(\widetilde{\mathcal{M}}) \\ &\}. \end{aligned}$$

Definition 12 (Model Tree)**.** *Given a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}} : \text{MOF}$, and a collection $\widetilde{M}$ of typed objects, such that $\widetilde{M} : \mathcal{M}$,*

$$(\widetilde{M}, \widetilde{\mathcal{M}}, <_c)$$

is called the model tree *of the model definition $\widetilde{M}$ iff*

(i) $\widetilde{M}$ is a set of typed objects that constitute the nodes in $tree(\widetilde{M}, \widetilde{\mathcal{M}})$.

(ii) $\widetilde{\mathcal{M}}$ is the metamodel definition that contains the definitions of the object types that are needed to define objects in $\widetilde{M}$.

[9]We assume that *isComposite* properties are always defined with an opposite property, the *containment property*, i.e., given a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}} : \text{MOF}$,

$$\forall \widetilde{p}\ (\widetilde{p} : \text{Property} \ \wedge\ \widetilde{p} \in \widetilde{\mathcal{M}} \ \wedge\ \widetilde{p}.isComposite = true \ \wedge\ \widetilde{p}.opposite(\widetilde{\mathcal{M}}) \neq \emptyset).$$

(iii) $<_c$ *is the partial function that defines the set of containment relationships in the model definition* $\widetilde{M}$ *as* $<_c (\widetilde{M}, \widetilde{\mathcal{M}})$.

(iv) $(\widetilde{M}, \leqslant_c)$ *is a partially ordered set, where* $\leqslant_c$ *is a binary relation defined by the transitive-reflexive closure of the relation that is defined by the function* $<_c$. *The relation* $\leqslant_c$ *is defined by the equation*

$$\leqslant_c = (<_c (\widetilde{M}, \widetilde{\mathcal{M}}))^*.$$

Given two objects $\widetilde{o_1}$ *and* $\widetilde{o_2}$ *such that* $\widetilde{o_1}, \widetilde{o_2} \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}}$ *and* $\widetilde{o_1}, \widetilde{o_2} \in \widetilde{M}$, *we obtain the equivalences*

$$(\widetilde{o_1} <_c \widetilde{o_2}) \iff (\widetilde{o_1}, \widetilde{o_2}) \in <_c (\widetilde{M}, \widetilde{\mathcal{M}})$$

and

$$(\widetilde{o_1} <_c^+ \widetilde{o_2}) \iff (\widetilde{o_1}, \widetilde{o_2}) \in (<_c (\widetilde{M}, \widetilde{\mathcal{M}}))^+,$$

where $(<_c (\widetilde{M}, \widetilde{\mathcal{M}}))^+$ *is the transitive closure of the relation that is defined by* $<_c (\widetilde{M}, \widetilde{\mathcal{M}})$.

(v) *The set*

$$\begin{aligned} root(\widetilde{M}, \widetilde{\mathcal{M}}) = \; & \{\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}} \mid \widetilde{M} : \mathcal{M} \wedge \widetilde{o} \in \widetilde{M} \wedge \\ & \nexists \widetilde{o}' (\widetilde{o}' \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}} \wedge \widetilde{o}' \in \widetilde{M} \wedge \widetilde{o} <_c^+ \widetilde{o}')\} \end{aligned}$$

is a singleton set.

Given a model tree $tree(\widetilde{M}, \widetilde{\mathcal{M}})$ as defined above, we can use the function

$$containments : T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}} \times [\![\mathcal{M}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \rightsquigarrow [\![\mathcal{M}_0]\!]_{\mathrm{MOF}}$$

to obtain the children nodes of a specific node in a model tree. This function is only defined for tuples $(\widetilde{o}, \widetilde{M}, \widetilde{\mathcal{M}})$, where $\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}}$, $\widetilde{M} : \mathcal{M}$, and $\widetilde{o} \in \widetilde{M}$ as follows:

$$containments(\widetilde{o}, \widetilde{M}, \widetilde{\mathcal{M}}) = \bigcup_{\widetilde{o}' \in \{\widetilde{o}' \mid \widetilde{o}' \in \widetilde{M} \wedge \widetilde{o}' <_c \widetilde{o}\}} (\ll \widetilde{o}' \gg).$$

where $\widetilde{o}' \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}}$ and the union operator is defined for configurations of objects, i.e., terms of sort $Configuration\{\mathcal{M}\}$, as follows:

$$\ll OC \gg \cup \ll OC' \gg = \ll OC \;\; OC' \gg,$$

where $OC, OC' : ObjectCollection\{\mathcal{M}\}$.

A model tree $tree(\widetilde{M}, \widetilde{\mathcal{M}})$ can also be viewed as a graph $G' = (V, E)$, where $V = \widetilde{M}$, and $E = <_c (\widetilde{M}, \widetilde{\mathcal{M}})$. Since $<_c (\widetilde{M}, \widetilde{\mathcal{M}}) \subseteq edges(\widetilde{M}, \widetilde{\mathcal{M}})$ for a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, the model tree $tree(\widetilde{M}, \widetilde{\mathcal{M}})$ is a subgraph of the model graph $graph(\widetilde{M}, \widetilde{\mathcal{M}})$, i.e.,

$$tree(\widetilde{M}, \widetilde{\mathcal{M}}) \subseteq graph(\widetilde{M}, \widetilde{\mathcal{M}}).$$

Note that a MOF tree $tree(\widetilde{\mathcal{M}_{\mathrm{MOF}}}, \widetilde{\mathrm{MOF}})$, such that $\widetilde{\mathcal{M}_{\mathrm{MOF}}} : \mathrm{MOF}$ and $\widetilde{\mathrm{MOF}} : \mathrm{MOF}$, is also a model tree.

The containment relation $<_c$ for a specific metamodel definition $\widetilde{\mathcal{M}}$ is reified at the metadata level, i.e., it is not represented by any algebraic structure in the MEL theory $reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$. However, this containment relation can be taken into account in computable functions in our MOF framework due to the formalization of the MOF Reflection facilities, which permit querying the metadata representation of types. Therefore, the containment relation $<_c$ can be used for both theoretical and practical purposes. The MOF Reflection facilities are discussed in detail in Section 8.

Definition 13 (Model subtree)**.** *Given two model definitions* $\widetilde{M}'$ *and* $\widetilde{M}$, *such that* $\widetilde{M}', \widetilde{M} : \mathcal{M}$, $(tree(\widetilde{M}', \widetilde{\mathcal{M}}), tree(\widetilde{M}, \widetilde{\mathcal{M}}))$ *is called a model subtree iff* $(tree(\widetilde{M}', \widetilde{\mathcal{M}}), tree(\widetilde{M}, \widetilde{\mathcal{M}}))$ *is a model subgraph, which is also denoted by* $tree(\widetilde{M}', \widetilde{\mathcal{M}}) \subseteq tree(\widetilde{M}, \widetilde{\mathcal{M}})$.

We define the function

$$allSuperClasses : [\![\mathrm{Class}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \rightarrow [\![\mathrm{MOF}_0]\!]_{\mathrm{MOF}}$$

to obtain the the collection of Class instances that are defined as super classes of a given Class instance $\widetilde{cl}$, which is defined in the metamodel definition $\widetilde{\mathcal{M}}$. Recall the union operator that is defined for configurations of objects above. The *allSuperClasses* function is defined by the mapping

$$
\begin{aligned}
&\mathit{allSuperClasses}(\widetilde{cl}, \widetilde{\mathcal{M}}) = \\
&\quad \mathit{ColToConf}(\widetilde{cl}.\mathit{superClass}(\widetilde{\mathcal{M}})) \cup \\
&\quad \bigcup_{\widetilde{cl}' \in \{\widetilde{cl}' : \textsc{Class} \mid \widetilde{cl}' \in \widetilde{cl}.\mathit{superClass}(\widetilde{\mathcal{M}})\}} (\mathit{allSuperClasses}(\widetilde{cl}', \widetilde{\mathcal{M}})),
\end{aligned}
$$

where the function

$$\mathit{ColToConf} : T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{Collection}\{\mathcal{M}\}} \to [\![\mathrm{MOF}_0]\!]_{\mathrm{MOF}}$$

maps a specific OCL collection of objects into a collection of typed objects of sort MOF_0.

Definition 14 (Root Object Type)**.** *In a metamodel definition* $\widetilde{\mathcal{M}}$*, a* root object type OT *is defined as an object type definition* $\widetilde{\mathrm{OT}}$*, such that* $\widetilde{\mathrm{OT}} : \mathrm{MOF}$ *and we have a containment* $\mathit{tree}(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}) \subseteq \mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$*, that is not referred to by any* Property *instance* $\widetilde{p}$ *such that*

$$
\begin{aligned}
&\widetilde{p} \in \widetilde{\mathcal{M}} && \text{(i)}\\
&\mathit{containment}(\widetilde{p}, \widetilde{\mathcal{M}}) = \mathit{true} && \text{(ii)}\\
&\widetilde{p}.\mathit{class}(\widetilde{\mathcal{M}}) \notin \\
&\quad \ll \mathit{root}(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}) \gg \ \cup\ \mathit{allSuperClasses}(\mathit{root}(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}), \widetilde{\mathcal{M}}) && \text{(iii)}\\
&\widetilde{p}.\mathit{type}(\widetilde{\mathcal{M}}) \in \\
&\quad \ll \mathit{root}(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}) \gg \ \cup\ \mathit{allSuperClasses}(\mathit{root}(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}), \widetilde{\mathcal{M}}) && \text{(iv)}
\end{aligned}
$$

We can define a partial function

$$\mathit{rootOT} : [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \rightsquigarrow [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$$

defined on metamodel definitions $\widetilde{\mathcal{M}}$ having a single root object type that provides the object type definition that is defined as root in the metamodel definition $\widetilde{\mathcal{M}}$. The rootOT function is defined by the mapping:

$$
\begin{aligned}
&\mathit{rootOT}(\widetilde{\mathcal{M}}) = \\
&\{\widetilde{\mathrm{OT}} \mid \widetilde{\mathrm{OT}} : \mathrm{MOF} \ \wedge\ \mathit{root}(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}) : \textsc{Class} \ \wedge \\
&\quad \nexists \widetilde{p}\ (\widetilde{p} : \textsc{Property} \ \wedge\ \widetilde{p} \in \widetilde{\mathcal{M}} \ \wedge \\
&\qquad \mathit{containment}(\widetilde{p}, \widetilde{\mathcal{M}}) = \mathit{true} \ \wedge \\
&\qquad \widetilde{p}.\mathit{class}(\widetilde{\mathcal{M}}) \notin \\
&\qquad\quad \ll \mathit{root}(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}) \gg \ \cup\ \mathit{allSuperClasses}(\mathit{root}(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}), \widetilde{\mathcal{M}}) \ \wedge \\
&\qquad \widetilde{p}.\mathit{type}(\widetilde{\mathcal{M}}) \in \\
&\qquad\quad \ll \mathit{root}(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}) \gg \ \cup\ \mathit{allSuperClasses}(\mathit{root}(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}), \widetilde{\mathcal{M}}) \\
&\quad) \\
&\}
\end{aligned}
$$

Under the assumption that there is only one root object type in a metamodel definition $\widetilde{\mathcal{M}}$, in a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, an instance of the object type that is defined by $\mathit{rootOT}(\widetilde{\mathcal{M}})$ becomes the root of the tree view $\mathit{tree}(\widetilde{M}, \widetilde{\mathcal{M}})$, i.e.,

$$\widetilde{\mathrm{OT}} = \mathit{rootOT}(\widetilde{\mathcal{M}}) \ \wedge\ \mathit{root}(\widetilde{M}, \widetilde{\mathcal{M}}) : \mathrm{OT}.$$

However, a model definition $\widetilde{M}$ can be viewed as a forest that is constituted by several trees. For example, the metamodel definition $\widetilde{\textsc{Book}}$ that is defined as a MOF class diagram in Fig. 6.8 permits defining the

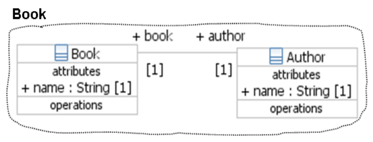


Figure 6.8: Metamodel definition with two root object types.

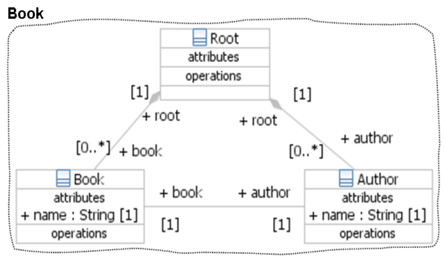


Figure 6.9: Metamodel definition with a single root object type.

following model definition[10]:

```
<<
    < oid#Book('Foo) : Book |
        name : "Tirant lo Blanch", author : oid#Author('Bar)
    >
    < oid#Author('Bar) : Author |
        name : "Joanot Martorell", book : oid#Book('Foo)
    >
>>,
```

where there are two roots. To simplify the semantics of a model type, we assume throughout that *there is always a single root object type in a metamodel definition* $\widetilde{\mathcal{M}}$.

When a metamodel definition $\widetilde{\mathcal{M}}$ has more than two root object types, a new object type can be created as a container for the root object type defintions. For example, the metamodel definition $\widetilde{\text{BOOK}}$ can be redefined as a rooted metamodel definition in Fig. 6.9 by adding the object type definition $\widetilde{\text{ROOT}}$.

Structure Definition

A model type $[\![\mathcal{M}]\!]_{\text{MOF}}$ is defined by the set of collections of typed objects that can be viewed as a model graph and as a model tree, i.e.,

$$\widetilde{M} : \mathcal{M} \implies \begin{array}{l} graph(\widetilde{M}, \widetilde{\mathcal{M}}) \textit{ is a model graph } \wedge \\ tree(\widetilde{M}, \widetilde{\mathcal{M}}) \textit{ is a model tree.} \end{array}$$

The $[\![\mathcal{M}]\!]_{\text{MOF}}$ type is defined by means of a membership axiom that is defined in the `EXT-MODEL{OBJ :: TH-OBJECT}` theory. This membership axiom is a generalization of the membership axiom that has been defined for the $\mathit{reflect}_{\text{MOF}}(\widetilde{\text{MOF}})$ theory in the `MOFSTRUCTURE` theory, in Section 6.2 and is defined as follows:

[10]Although we have not seen yet how instances of an object type can be defined, the example can be understood once the $\mathit{reflect}_{\text{MOF}}(\widetilde{\text{MOF}})$ theory is understood. In this section, we present how a term of this kind can be defined by using the $\mathit{reflect}_{\text{MOF}}(\widetilde{\text{BOOK}})$ theory.

$$\boxed{\left.\begin{array}{l} \widetilde{M} : \mathcal{M}_0 \ \wedge \ \widetilde{\mathcal{M}} : \mathrm{MOF} \ \wedge \\ \mathit{noBrokenLinks}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{true} \ \wedge \\ \mathit{wellTypedLinks}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{true} \ \wedge \\ \mathit{singleContainer}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{true} \ \wedge \\ \mathit{singleRoot}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{true} \ \wedge \\ \mathit{validProperties}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{true} \ \wedge \\ \mathit{allProperties}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{true} \end{array}\right\} \Longrightarrow \widetilde{M} : \mathcal{M}}$$

To define the functions that constitute the conditions of the membership, we first consider some previous definitions. *SpecMEL* is the data type of *finitely-presented* MEL theories, that is, theories of the form $(S, <, \Omega, E \cup A)$, where all the components are finite. Without loss of generality we assume countable sets *Sorts* and *OpNames*, so that each set of sorts S is a finite subset of *Sorts*, and the operator names in Ω are a finite subset of *OpNames*. To obtain the sort that corresponds to a class, we define the function *ClassSort* : Class $\rightarrow$ *Sorts*, which is defined for MOF Class instances as follows: *ClassSort* : $\widetilde{cl} \mapsto \widetilde{cl}.\mathit{name}$, where $(\widetilde{cl} : \textsc{Class})$.

The set of types D' that can be used to define a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, is defined by the equation

$$\begin{array}{l} \mathrm{D}' = \\ \quad \{\ \texttt{Bool, String, Int, Float, Oid, Object\#MOF}\ \} \cup \\ \quad \{(\widetilde{enum}.\mathit{name}) \mid \widetilde{enum} : \textsc{Enumeration} \ \wedge \ \widetilde{enum} \in \widetilde{\mathcal{M}}\}, \end{array}$$

where we take into account the enumeration types that are defined in $\widetilde{\mathcal{M}}$. The set D is defined by the equation

$$\begin{array}{ll} \mathrm{D} = & \bigcup_{\mathrm{T} \in \mathrm{D}'} \ \{\ \texttt{NeSet\{T\}, Set\{T\}, NeOrderedSet\{T\}, OrderedSet\{T\},} \\ & \qquad\qquad \texttt{NeBag\{T\}, Bag\{T\}, NeSequence\{T\}, Sequence\{T\},} \\ & \qquad\qquad \texttt{Collection\{T\}}\} \\ & \cup\ \mathrm{D}', \end{array}$$

The set of values that can be used to define object properties in a specific object $\widetilde{o}$, such that $\widetilde{o} \in \widetilde{M}$ is defined as

$$\mathcal{D} = \bigcup_{t \in D} (T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), t}).$$

Given a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, and an object $\widetilde{o_1}$, such that $T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{Object\#}\mathcal{M}}$ and $\widetilde{o_1} \in \widetilde{M}$, and a Property instance $\widetilde{p}$, such that $\widetilde{p} : \textsc{Property}$, $\widetilde{p} : \widetilde{\mathcal{M}}$ and $\mathit{class}(\widetilde{o_1}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}), \mathit{ClassSort}(\widetilde{p}.\mathit{class}(\widetilde{\mathrm{MOF}}))}$, the expression $\widetilde{o_1}.(\widetilde{p}.\mathit{name})$ obtains the value of the property $\widetilde{p}$ in $\widetilde{o_1}$. For example, consider $\widetilde{\mathrm{MOF}}$ as both the model definition $\widetilde{M}$ and the metamodel definition $\widetilde{\mathcal{M}}$, an object $\widetilde{o_1}$, which is defined by the equation

$$\widetilde{o_1} = \texttt{< oid\#Class('Foo) : Class | name : "Property",... >}$$

in the model definition $\widetilde{\mathrm{MOF}}$, and the Property instance $\widetilde{p}$, which is defined by the equation

$$\widetilde{p} = \texttt{< oid\#Property('Bar) : Property | name : "name",...>}$$

that belongs to the NamedElement object type in the metamodel definition $\widetilde{\mathrm{MOF}}$. The expression $\widetilde{o_1}.(\widetilde{p}.\mathit{name})$ corresponds to the term $\widetilde{o_1}$`.name`, which is reduced to the value `"Property"`.

Taking these considerations into account, the functions that constitute the conditions of the membership that defines the type $[\![\mathcal{M}]\!]_{\mathrm{MOF}}$ are:

- *noBrokenLinks:* checks that there are no broken edges in the graph $\mathit{graph}(\widetilde{M}, \widetilde{\mathcal{M}})$, i.e., an object in $\widetilde{M}$ does not refer to an undefined object by means of an object-typed property.

$$\mathit{noBrokenLinks} : [\![\mathcal{M}_0]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow [\![\textsc{Boolean}]\!]_{\mathrm{MOF}}$$

```
$\mathit{noBrokenLinks}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{false}$
    $\mathit{if}\ \exists \widetilde{o_1} (\widetilde{o_1} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{Object\#\mathcal{M}}} \ \wedge$
        $\widetilde{M} : \mathcal{M}_0 \ \wedge\ \widetilde{\mathcal{M}} : \mathrm{MOF} \ \wedge\ \widetilde{o_1} \in \widetilde{M} \ \wedge$
        $\exists \widetilde{p}\ (\widetilde{p} : \textsc{Property} \ \wedge\ \widetilde{p} \in \widetilde{\mathcal{M}} \ \wedge\ \widetilde{p}.\mathit{type}(\widetilde{\mathcal{M}}) : \textsc{Class} \ \wedge$
          $\mathit{class}(\widetilde{o_1}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{ClassSort}(\widetilde{p}.\mathit{class}(\widetilde{\mathcal{M}}))} \ \wedge\ \widetilde{o_1}.(\widetilde{p}.\mathit{name}) \neq \emptyset \ \wedge$
          $\exists \mathit{OID} (\mathit{OID} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{Oid\#\mathcal{M}}} \ \wedge\ \mathit{OID} \in \widetilde{o_1}.(\widetilde{p}.\mathit{name}) \ \wedge$
            $\forall \widetilde{o_2} (\widetilde{o_2} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{Object\#\mathcal{M}}} \ \wedge\ \widetilde{o_2} \in \widetilde{M} \rightarrow \mathit{oid}(\widetilde{o_2}) \neq \mathit{OID})$
          )
        )
      )
$\mathit{noBrokenLinks}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{true} \quad \mathit{otherwise}$
```

- *wellTypedLinks:* checks that the nodes of a graph $\mathit{graph}(\widetilde{M}, \widetilde{\mathcal{M}})$, which are objects in $\widetilde{M}$, are typed with the object types that participate in the corresponding property definition in $\widetilde{\mathcal{M}}$.

$$\mathit{wellTypedLinks} : [\![\mathcal{M}_0]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow [\![\textsc{Boolean}]\!]_{\mathrm{MOF}}$$

```
$\mathit{wellTypedLinks}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{false}$
  $\mathit{if}\ \exists (\widetilde{o_1}, \widetilde{o_2})\ (\widetilde{o_1}, \widetilde{o_2} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{Object\#\mathcal{M}}} \ \wedge\ \widetilde{o_1}, \widetilde{o_2} \in \widetilde{M} \ \wedge$
    $\widetilde{M} : \mathcal{M}_0 \ \wedge\ \widetilde{\mathcal{M}} : \mathrm{MOF} \ \wedge\ \exists \widetilde{p}\ (\widetilde{p} : \textsc{Property} \ \wedge\ \widetilde{p} \in \widetilde{\mathcal{M}} \ \wedge\ \widetilde{p}.\mathit{type}(\widetilde{\mathcal{M}}) : \textsc{Class} \ \wedge$
      $\mathit{class}(\widetilde{o_1}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{ClassSort}(\widetilde{p}.\mathit{class}(\widetilde{\mathcal{M}}))} \ \wedge\ \widetilde{o_2} \in \widetilde{o_1}.(\widetilde{p}.\mathit{name})(\widetilde{M}) \ \wedge$
      $\mathit{class}(\widetilde{o_2}) \notin T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{ClassSort}(\widetilde{p}.\mathit{type}(\widetilde{\mathcal{M}}))}$
    )
  )
$\mathit{wellTypedLinks}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{true} \quad \mathit{otherwise}$
```

- *singleContainer*: checks that an object in a tree $\mathit{tree}(\widetilde{M}, \widetilde{\mathcal{M}})$ cannot be contained in two different objects.

$$\mathit{singleContainer} : [\![\mathcal{M}_0]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow [\![\textsc{Boolean}]\!]_{\mathrm{MOF}}$$

$$
\begin{array}{l}
\mathit{singleContainer}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{false} \\
\quad \mathit{if}\ \exists \widetilde{o_1}, \widetilde{o_2}\ (\widetilde{o_1}, \widetilde{o_2} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{Object}\#\mathcal{M}}\ \wedge\ \widetilde{o_1}, \widetilde{o_2} \in \widetilde{M}\ \wedge\ \widetilde{o_1} \neq \widetilde{o_2}\ \wedge \\
\quad\quad \exists \widetilde{o_3}\ (\widetilde{o_3} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{Object}\#\mathcal{M}}\ \wedge\ \widetilde{o_3} \in \widetilde{M}\ \wedge\ \widetilde{o_3} \neq \widetilde{o_1}\ \wedge\ \widetilde{o_3} \neq \widetilde{o_2}\ \wedge \\
\quad\quad\quad \widetilde{o_3} <_c^+ \widetilde{o_1}\ \wedge\ \widetilde{o_3} <_c^+ \widetilde{o_2} \\
\quad\quad) \\
\quad) \\
\mathit{singleContainer}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) = \mathit{true} \quad \mathit{otherwise}
\end{array}
$$

- *singleRoot:* checks that a tree $\mathit{tree}(\widetilde{M}, \widetilde{\mathcal{M}})$ has a single root object.

$$\mathit{singleRoot} : [\![\mathcal{M}_0]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow [\![\mathrm{Boolean}]\!]_{\mathrm{MOF}}$$

$$
\begin{array}{l}
\mathit{singleRoot}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{false} \\
\quad \mathit{if}\ \exists \widetilde{o_1}\ (\widetilde{o_1} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{Object}\#\mathcal{M}}\ \wedge\ \widetilde{\mathcal{M}} : \mathrm{MOF}\ \wedge\ \widetilde{o_1} \in \widetilde{M}\ \wedge \\
\quad\quad \nexists \widetilde{o_2} (\widetilde{o_2} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{Object}\#\mathcal{M}}\ \wedge\ \widetilde{o_2} \in \widetilde{M}\ \wedge\ \widetilde{o_1} <_c^+ \widetilde{o_2})\ \wedge \\
\quad\quad \exists \widetilde{o_3}\ (\widetilde{o_3} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{Object}\#\mathcal{M}}\ \wedge\ \widetilde{o_3} \in \widetilde{M}\ \wedge\ \widetilde{o_3} \neq \widetilde{o_1}\ \wedge \\
\quad\quad\quad \nexists \widetilde{o_4} (\widetilde{o_4} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \mathit{Object}\#\mathcal{M}}\ \wedge\ \widetilde{o_4} \in \widetilde{M}\ \wedge\ \widetilde{o_3} <_c^+ \widetilde{o_4}) \\
\quad) \\
\mathit{singleRoot}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{true} \quad \mathit{otherwise}
\end{array}
$$

- *validProperties:* checks that the property values that are defined for a specific object $\widetilde{o}$ in a model definition $\widetilde{M}$ are defined by means of properties that belong to the object type of the object $\widetilde{o}$, or to any of its supertypes.

$$\mathit{validProperties} : [\![\mathcal{M}_0]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow [\![\mathrm{Boolean}]\!]_{\mathrm{MOF}}$$

$$
\begin{array}{l}
\mathit{validProperties}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{false} \\
\\
\mathit{if}\ \exists(\widetilde{o})\ (\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\, \mathit{Object}\#\mathcal{M}}\ \wedge\ \widetilde{o} \in \widetilde{M}\ \wedge \\
\quad (\\
\qquad \exists(\mathit{prop} : \mathit{value})(\mathit{prop} \in \mathit{OpNames}\ \wedge\ \mathit{value} \in \mathcal{D}\ \wedge \\
\\
\qquad\quad (\mathit{prop} : \mathit{value}) \in \mathit{getProperties}(\widetilde{o})\ \wedge\ \forall \widetilde{p}(\widetilde{p} : \textsc{Property}\ \wedge\ \widetilde{p} \in \widetilde{\mathcal{M}}\ \wedge \\
\\
\qquad\qquad \mathit{class}(\widetilde{o}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\, \mathit{ClassSort}(\widetilde{p}.\mathit{class}(\widetilde{\mathcal{M}}))}\ \rightarrow\ \widetilde{p}.\mathit{name} \neq \mathit{prop} \\
\qquad\quad) \\
\qquad) \\
\quad \vee \\
\qquad \exists \mathit{prop}(\mathit{prop} \in \mathit{OpNames}\ \wedge\ \mathit{prop} \in \mathit{getProperties}(\widetilde{o})\ \wedge \\
\\
\qquad\quad \forall \widetilde{p}(\widetilde{p} : \textsc{Property}\ \wedge\ \widetilde{p} \in \widetilde{\mathcal{M}}\ \wedge \\
\\
\qquad\qquad \mathit{class}(\widetilde{o}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\, \mathit{ClassSort}(\widetilde{p}.\mathit{class}(\widetilde{\mathcal{M}}))}\ \rightarrow\ \widetilde{p}.\mathit{name} \neq \mathit{prop} \\
\qquad\quad) \\
\qquad) \\
\quad) \\
\mathit{validProperties}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{true} \quad \mathit{otherwise}
\end{array}
$$

- *allProperties:* checks that every object $\widetilde{o}$ in a model definition $\widetilde{M}$ contains a property value for each one of the properties that are defined for the object type of $\widetilde{o}$ or any of its supertypes.

$$\mathit{allProperties} : [\![\mathcal{M}_0]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow [\![\textsc{Boolean}]\!]_{\mathrm{MOF}}$$

$$
\begin{array}{l}
\mathit{allProperties}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{false} \\
\\
\mathit{if}\ \exists(\widetilde{o})\ (\widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\, \mathit{Object}\#\mathcal{M}}\ \wedge\ \widetilde{o} \in \widetilde{M}\ \wedge \\
\\
\exists \widetilde{p}(\widetilde{p} : \textsc{Property}\ \wedge\ \widetilde{p} \in \widetilde{\mathcal{M}}\ \wedge\ \mathit{class}(\widetilde{o}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\, \mathit{ClassSort}(\widetilde{p}.\mathit{class}(\widetilde{\mathcal{M}}))}\ \wedge \\
\quad (\\
\qquad \nexists(\mathit{prop} : \mathit{value})(\mathit{prop} \in \mathit{OpNames}\ \wedge\ \mathit{value} \in \mathcal{D}\ \wedge \\
\\
\qquad\quad \widetilde{p}.\mathit{name} = \mathit{prop}\ \wedge\ (\mathit{prop} : \mathit{value}) \in \mathit{getProperties}(\widetilde{o}) \\
\qquad) \\
\quad \wedge \\
\qquad \nexists(\mathit{prop})(\mathit{prop} \in \mathit{OpNames}\ \wedge\ \mathit{value} \in \mathcal{D}\ \wedge \\
\\
\qquad\quad \widetilde{p}.\mathit{name} = \mathit{prop}\ \wedge\ (\mathit{prop}) \in \mathit{getProperties}(\widetilde{o}) \\
\qquad) \\
\quad) \\
) \\
\mathit{allProperties}(\widetilde{M}, \widetilde{\mathcal{M}}) = \mathit{true} \quad \mathit{otherwise}
\end{array}
$$

The `EXT-MODEL{OBJ :: TH-OBJECT}` theory provides the following types:

(i) $\mathcal{M}_0$, which is represented by the sort *Configuration*$\{\mathcal{M}\}$ in the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory, is the type of collections of typed objects that have no structure. The semantics of the $\mathcal{M}_0$ is defined by the equation

$$\boxed{[\![\mathcal{M}_0]\!]_{\mathrm{MOF}} = T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\, \mathit{Configuration}\{\mathcal{M}\}},}$$

where

$$\boxed{\widetilde{M} : \mathcal{M}_0 \Leftrightarrow \widetilde{M} \in [\![\mathcal{M}_0]\!]_{\mathrm{MOF}};}$$

(ii) $\mathcal{M}$, which is represented by *ModelType*$\{\mathcal{M}\}$ in the *reflect*$_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory, is the type of collections of typed objects that keep both a graph and a tree structure. The semantics of the $\mathcal{M}$ type is defined by the equation

$$\boxed{[\![\mathcal{M}]\!]_{\mathrm{MOF}} = T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\mathit{ModelType}\{\mathcal{M}\}},}$$

and the structural conformance relation between a model definition $\widetilde{M} : \mathcal{M}$ and its corresponding model type $\mathcal{M}$ is then formally defined by the equivalence

$$\boxed{\widetilde{M} : \mathcal{M} \Leftrightarrow \widetilde{M} \in [\![\mathcal{M}]\!]_{\mathrm{MOF}};}$$

(iii) and $(\mathcal{M}, \mathcal{C})$, which is represented by *ConsistentModelType*$\{\mathcal{M}\}$ in the *reflect*$_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory, is the type of model types that conform to $\mathcal{M}$ and that satisfy the OCL constraints $\mathcal{C}$. The semantics of the $(\mathcal{M}, \mathcal{C})$ type is defined by the equation

$$\boxed{[\![(\mathcal{M}, \mathcal{C})]\!]_{\mathrm{MOF}} = T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\mathit{ConsistentModelType}\{\mathcal{M}\}},}$$

where

$$\boxed{\widetilde{M} : (\mathcal{M}, \mathcal{C}) \Leftrightarrow \widetilde{M} \in [\![(\mathcal{M}, \mathcal{C})]\!]_{\mathrm{MOF}}.}$$

The domain $T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\mathit{ConsistentModelType}\{\mathcal{M}\}}$ is defined in Section 7.

6.3.2 Specific Semantics of a Metamodel Definition $\widetilde{\mathcal{M}}$

The function

$$\mathit{reflect}_{\mathrm{MOF}} : [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \to \mathit{SpecMEL}$$

maps each object, which is an instance of a specific object type of *reflect*$_{\mathrm{MOF}}(\widetilde{MOF})$, of a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF, to a theory $(S, <, \Omega, E \cup A)$. Given a metamodel definition $\widetilde{\mathcal{M}}$, the resulting theory *reflect*$_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ provides the algebraic semantics for each of the types that are defined as collections $\widetilde{T}$, such that $\widetilde{T}$: MOF, of objects in $\widetilde{\mathcal{M}}$, such that $\mathit{tree}(\widetilde{T}, \widetilde{\mathrm{MOF}}) \subseteq \mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$. A type definition $\widetilde{T}$ can be a definition of a model type, of a package, of a primitive type, of an enumeration type, or of an object type. When $\widetilde{\mathcal{M}} = \widetilde{\mathrm{MOF}}$, the *reflect*$_{\mathrm{MOF}}$ function is defined as

$$\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}) = \texttt{MODEL\{MOF\}} \;\cup\; \texttt{MOFSTRUCTURE}.$$

When $\widetilde{\mathcal{M}} \neq \widetilde{\mathrm{MOF}}$, the *reflect*$_{\mathrm{MOF}}$ function is defined for a given metamodel definition $\widetilde{\mathcal{M}}$ as

$$\mathit{reflect}_{\texttt{MOF}}(\widetilde{\mathcal{M}}) = \texttt{EXT-MODEL}\{\mathcal{M}\},$$

where $\mathcal{M}$ is the view that maps the `TH-OBJECT` theory, which represents the formal parameter of the `EXT-MODEL{OBJ :: TH-OBJECT}` theory, to the theory that is generated from $\widetilde{\mathcal{M}}$ by means of the function *defineParameter*.

While the `EXT-MODEL{OBJ :: TH-OBJECT}` theory provides the generic semantics for the types of any metamodel definition $\widetilde{\mathcal{M}}$, the *defineParameter* function provides the semantics for the types that are provided in a specific metamodel definition $\widetilde{\mathcal{M}}$, i.e., enumeration type definitions and object type definitions. The semantics of the function

$$\mathit{defineParameter} : [\![\textsc{NamedElement}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow \mathit{SpecMEL}$$

is defined compositionally by using the theory union operator

$$\begin{array}{ll} \_\cup\_: & ((S, <, \Omega, E \cup A),\;\; (S', <', \Omega', E' \cup A')) \;\;\mapsto \\ & (S \cup S', < \cup <', \Omega \cup \Omega', (E \cup E') \cup (A \cup A')), \end{array}$$

which is well-defined provided that the reflexive-transitive closure of the relation $< \cup <'$, which is understood as the set-theoretic union of the relations $<$ and $<'$, is a partial order on $S \cup S'$.

The *defineParameter* function traverses the objects that constitute a metamodel definition $\widetilde{\mathcal{M}}$ by means of the containment relation $<_c$ $(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}})$. More specifically, the objects that are the root objects in a specific type definition in $\widetilde{\mathcal{M}}$ are mapped to elements in a MEL theory, providing the algebraic representation of the corresponding type definition. Given the root package $\widetilde{rootPk}$ of the metamodel definition $\widetilde{\mathcal{M}}$, where $\widetilde{rootPk}$: PACKAGE and $\widetilde{rootPk} = root(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}})$, the resulting $defineParameter(\widetilde{rootPk}, \widetilde{\mathcal{M}})$ theory constitutes the theory that is mapped to the `TH-OBJECT` theory by means of the view $\mathcal{M}$.

The metamodel definition $\widetilde{\text{MOF}}$ provides the type definitions that constitute the MOF meta-metamodel. The specification $defineParameter(root(\widetilde{\text{MOF}}, \widetilde{\text{MOF}}), \widetilde{\text{MOF}})$ provides the `mod#MOF` theory that has been defined in an adhoc way in Section 6.2. Note that this adhoc definition is necessary because the $[\![\text{MOF}]\!]_{\text{MOF}}$ domain is used to define the *defineParameter* function.

In this section, the *defineParameter* function is defined for any metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF and $\widetilde{\mathcal{M}} \neq \widetilde{\text{MOF}}$, providing the algebraic semantics for each one of the enumeration and object types that can be defined in $\widetilde{\mathcal{M}}$, and a formal definition of the *isValueOf* relation between each value that can appear in a model definition $\widetilde{M} : \mathcal{M}$ and its corresponding algebraic type in $reflect_{\text{MOF}}(\widetilde{\mathcal{M}})$.

Package

The nature of the Package construct is merely syntactical, providing a namespace for its contained elements, so that different types can be defined as data with the same name in different packages. Therefore, the semantics of a Package definition $\widetilde{\text{PK}}$, such that $\widetilde{\text{PK}}$: MOF, $root(\widetilde{\text{PK}}, \widetilde{\text{MOF}})$: PACKAGE, and $tree(\widetilde{\text{PK}}, \widetilde{\text{MOF}}) \subseteq tree(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}})$, is regarded as a MEL theory that constitutes a syntactical unit for the sorts and operators that are generated for its contained types.

Each metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF, contains a PACKAGE instance $\widetilde{pk}$, such that $\widetilde{pk} = root(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}})$, that constitutes a root element in the metamodel definition $\widetilde{\mathcal{M}}$. The root package $\widetilde{pk}$ may or may not contain other *nested packages*, providing the types that can be used to define a model $\widetilde{M} : \mathcal{M}$. The name of the root package is used to qualify the symbols of the generic sorts that are needed to define objects in a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$. When a PACKAGE instance $\widetilde{pk}$ satisfies the condition $\widetilde{pk} = root(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}})$, the *defineParameter* function is given by the following equation:

$$
\begin{aligned}
& defineParameter(\widetilde{pk}, \widetilde{\mathcal{M}}) = \\
& \quad (S, <, \Omega, E \cup A) \cup \\
& \quad \bigcup_{\widetilde{pk}' \in \{\widetilde{pk}' \,:\, \text{PACKAGE} \mid \widetilde{pk}' \in \widetilde{pk}.nestedPackage(\widetilde{\mathcal{M}})\}} defineParameter(\widetilde{pk}', \widetilde{\mathcal{M}}) \;\cup \\
& \quad \bigcup_{\widetilde{t} \in \{\widetilde{t} \,:\, \text{TYPE} \mid \widetilde{t} \in \widetilde{pk}.ownedType(\widetilde{\mathcal{M}})\}} defineParameter(\widetilde{t}, \widetilde{\mathcal{M}}).
\end{aligned}
$$

where the symbols of generic sorts and constructors are qualified with the name of the root package as follows:

$$\begin{aligned}
S = \{&\mathit{Object}\#\widetilde{pk}.\mathit{name},\ \mathit{Oid}\#\widetilde{pk}.\mathit{name},\ \mathit{Cid}\#\widetilde{pk}.\mathit{name},\\
&\mathit{Property}\#\widetilde{pk}.\mathit{name},\ \mathit{PropertySet}\#\widetilde{pk}.\mathit{name}\}\\
< = \{&(\mathit{Property}\#\widetilde{pk}.\mathit{name} < \mathit{PropertySet}\#\widetilde{pk}.\mathit{name}),\\
&(\mathit{Oid}\#\widetilde{pk}.\mathit{name} < \mathit{Oid})\}\\
\Omega = \{&(\mathit{noneProperty}\#\widetilde{pk}.\mathit{name} :\rightarrow \mathit{Property}\#\widetilde{pk}.\mathit{name}),\\
&(\_,\_ : \mathit{PropertySet}\#\widetilde{pk}.\mathit{name} \times \mathit{PropertySet}\#\widetilde{pk}.\mathit{name}\\
&\quad \rightarrow \mathit{PropertySet}\#\widetilde{pk}.\mathit{name}\ [\mathit{assoc\ comm\ id} : \mathit{noneProperty}\#\widetilde{pk}.\mathit{name}]),\\
&(<\_:\_|\_>: \mathit{Oid}\#\widetilde{pk}.\mathit{name} \times \mathit{Cid}\#\widetilde{pk}.\mathit{name} \times \mathit{PropertySet}\#\widetilde{pk}.\mathit{name}\\
&\quad \rightarrow \mathit{Object}\#\widetilde{pk}.\mathit{name}[\mathit{object}]),\\
&(\mathit{oid}\ :\ \mathit{Object}\#\widetilde{pk}.\mathit{name}\ \rightarrow\ \mathit{Oid}\#\widetilde{pk}.\mathit{name})\\
&(\mathit{class}\ :\ \mathit{Object}\#\widetilde{pk}.\mathit{name}\ \rightarrow\ \mathit{Cid}\#\widetilde{pk}.\mathit{name})\\
&(\mathit{properties}\ :\ \mathit{Object}\#\widetilde{pk}.\mathit{name}\ \rightarrow\ \mathit{PropertySet}\#\widetilde{pk}.\mathit{name}\}\\
E = \{&(\mathit{oid}\ : <\mathit{OID} : \mathit{CID} \mid \mathit{PS} > =\ \mathit{OID}),\\
&(\mathit{class}\ : <\mathit{OID} : \mathit{CID} \mid \mathit{PS} > =\ \mathit{CID}),\\
&(\mathit{properties}\ : <\mathit{OID} : \mathit{CID} \mid \mathit{PS} > =\ \mathit{PS})\}
\end{aligned}$$

When the PACKAGE instance $\widetilde{pk}$ is not the root element of the metamodel definition $\widetilde{\mathcal{M}}$, i.e., $\widetilde{pk} \neq \mathit{root}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$, the *defineParameter* function is defined as follows:

$$\mathit{defineParameter}(\widetilde{pk}, \widetilde{\mathcal{M}}) =$$

$$\bigcup_{\widetilde{pk}' \in \{\widetilde{pk}' : \mathrm{PACKAGE} \mid \widetilde{pk}' \in \widetilde{pk}.\mathit{nestedPackage}(\widetilde{\mathcal{M}})\}} \mathit{defineParameter}(\widetilde{pk}', \widetilde{\mathcal{M}}) \ \cup$$

$$\bigcup_{\widetilde{t} \in \{\widetilde{t} : \mathrm{TYPE} \mid \widetilde{t} \in \widetilde{pk}.\mathit{ownedType}(\widetilde{\mathcal{M}})\}} \mathit{defineParameter}(\widetilde{t}, \widetilde{\mathcal{M}}).$$

The constructor that permits the definition of terms of sort $\mathit{Object}\#\widetilde{pk}.\mathit{name}$ is

$$\begin{aligned}
<\_:\_|\_> :\quad & \mathit{Oid}\#\widetilde{pk}.\mathit{name} \times \mathit{Cid}\#\widetilde{pk}.\mathit{name} \times \mathit{PropertySet}\#\widetilde{pk}.\mathit{name}\\
& \rightarrow\ \mathit{Object}\#\widetilde{pk}.\mathit{name},
\end{aligned}$$

where the the first argument is an object identifier, the second argument is a class name and the third argument is a multiset of comma-separated pairs of property values. Three operators are defined to project the contents of an instance: *oid*, *class* and *properties*. In addition, $\mathit{defineParameter}(\widetilde{pk}, \widetilde{\mathcal{M}})$ sets the name of the theory that represents the actual parameter for the `EXT-MODEL{OBJ :: TH-OBJECT}` theory as $\mathit{mod}\#\widetilde{pk}.\mathit{name}$.

As example, we consider the metamodel definition $\widetilde{\mathrm{RDBMS}} : \mathrm{MOF}$, shown in Fig. 2.2. The $\mathit{defineParameter}(\mathit{root}(\widetilde{\mathrm{RDBMS}}, \widetilde{\mathrm{MOF}}), \widetilde{\mathrm{RDBMS}})$ theory is specified, in Maude notation, as follows:

```
mod mod#rdbms is
    sorts Object#rdbms Oid#rdbms Cid#rdbms Property#rdbms
    PropertySet#rdbms .

    subsort Property#rdbms < PropertySet#rdbms .
    subsort Oid#rdbms < Oid .

    op noneProperty#rdbms : -> Property#rdbms [ctor] .
    op _‘,_ : PropertySet#rdbms PropertySet#rdbms -> PropertySet#rdbms
        [ctor assoc comm id: noneProperty#rdbms] .
    op <_:_|_> : Oid#rdbms Cid#rdbms PropertySet#rdbms ->
        Object#rdbms [ctor] .
    op oid : Object#rdbms -> Oid#rdbms .
    eq oid(< OID:Oid#rdbms : CID:Cid#rdbms | PS:PropertySet#rdbms >) =
        OID:Oid#rdbms .

    op class : Object#rdbms -> Cid#rdbms .
    eq class(< OID:Oid#rdbms : CID:Cid#rdbms | PS:PropertySet#rdbms >) =
        CID:Cid#rdbms .

    op properties : Object#rdbms -> PropertySet#rdbms .
    eq properties(< OID:Oid#rdbms : CID:Cid#rdbms |
        PS:PropertySet#rdbms >) = PS:PropertySet#rdbms .
    ...
endfm
```

We usually refer to sorts that are qualified with package information, such as $Object\#\widetilde{pk}.name$, by means of the the symbol $\mathcal{M}$, i.e., $Object\#\mathcal{M}$.

Enumeration Types

$\widetilde{\mathrm{ET}}$, such that $\widetilde{\mathrm{ET}}$: MOF, is called an enumeration type definition in the metamodel definition $\widetilde{\mathcal{M}}$ iff it satisfies the following conditions:

$$root(\widetilde{\mathrm{ET}}, \widetilde{\mathrm{MOF}}) : \textsc{Enumeration} \ \wedge \ tree(\widetilde{\mathrm{ET}}, \widetilde{\mathrm{MOF}}) \subseteq tree(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}).$$

For example, the enumeration in the metamodel definition $\widetilde{\mathrm{RDBMS}}$: MOF, shown in Fig. 2.2, is defined as the following collection $\widetilde{\mathrm{RD\textsc{ata}T\textsc{ype}}}$: MOF of objects:

```
< EnumOID : Enumeration |
    name : "RDataType", literal : literalsCol, EnumPS >
< LiteralOID1 : EnumerationLiteral |
    name : "VARCHAR", enumeration : EnumOID, LiteralPS1 >
< LiteralOID2 : EnumerationLiteral |
    name : "NUMBER", enumeration : EnumOID, LiteralPS2 >
< LiteralOID3 : EnumerationLiteral |
    name : "BOOLEAN", enumeration : EnumOID, LiteralPS3 >
< LiteralOID4 : EnumerationLiteral |
    name : "DATE", enumeration : EnumOID, LiteralPS4 >
< LiteralOID5 : EnumerationLiteral |
    name : "DECIMAL", enumeration : EnumOID, LiteralPS5 >
```

where `EnumOID : oid#Enumeration`, `LiteralOID1, LiteralOID2, LiteralOID3, LiteralOID4 and LiteralOID5 : oid#EnumerationLiteral`, `EnumPS, LiteralPS1, LiteralPS2, LiteralPS3, LiteralPS4 and LiteralPS5 : PropertySet#rdbms`, `literalsCol` is a collections of object identifiers, and `LiteralOID1, LiteralOID2, LiteralOID3, LiteralOID4, LiteralOID5` $\in$ `literalsCol`.

Enumeration types are represented in the $reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory as sorts whose carrier is constituted by a finite set of constants. These sorts and constants are defined by means of the function

$$defineParameter_{Enum} : [\![\textsc{Enumeration}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow SpecMEL,$$

which maps an Enumeration instance to a MEL theory as follows:

$$\begin{array}{l} defineParameter_{Enum}(\widetilde{enum}, \widetilde{\mathcal{M}}) = \\ \quad (\{\widetilde{enum}.name\}, \\ \quad \emptyset, \\ \quad \{(\widetilde{l}.name \ :\to \ \widetilde{enum}.name) \mid \widetilde{l} : \textsc{EnumerationLiteral} \ \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad \widetilde{l} \in \widetilde{enum}.ownedLiteral(\widetilde{\mathcal{M}})\}, \\ \quad \emptyset). \end{array}$$

where $\widetilde{enum}$: Enumeration, and $\widetilde{\mathcal{M}}$: MOF. The enumeration type that is defined in the relational metamodel definition $\widetilde{\mathrm{RDBMS}}$, shown in Fig. 2.2, is algebraically represented as a MEL theory, which is presented in Maude notation as

```
fmod mod#RDataType is
    sort RDataType .
    op VARCHAR : -> RDataType .
    op NUMBER : -> RDataType .
    op BOOLEAN : -> RDataType .
    op DATE : -> RDataType .
    op DECIMAL : -> RDataType .
endfm
```

Given the set *ViewNames* of view names that can be defined for parameterized MEL theories in *SpecMEL*, to enable the definition of collections of literals of a specific enumeration type in the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory, the *defineParameter* function is defined for ENUMERATION instances as follows:

$$\mathit{defineParameter}(\widetilde{enum}, \widetilde{\mathcal{M}}) = \texttt{OCL-COLLECTION-TYPES}\{\mathit{getEnumViewName}(\widetilde{enum})\},$$

where the function $\mathit{getEnumViewName} : [\![\mathrm{ENUMERATION}]\!]_{\mathrm{MOF}} \longrightarrow \mathit{ViewNames}$ obtains the name of the view that maps the TRIV theory to the $\mathit{defineParameter}_{Enum}(\widetilde{enum}, \widetilde{\mathcal{M}})$ theory. The *getEnumViewName* is defined by means of the equation

$$\mathit{getEnumViewName}(\widetilde{enum}) = \widetilde{enum}.\mathit{name}.$$

For the example, this view is defined, in Maude notation, as follows

```
view RDataType from TRIV to mod#RDataType is
    sort Elt to RDataType .
endv
```

The algebraic semantics of an enumeration type ET is defined using the initial algebra of $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ as

$$\boxed{[\![\mathrm{ET}]\!]_{\mathrm{MOF}} = T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), \widetilde{enum}.name}}$$

where $\widetilde{enum}$: ENUMERATION, and $\widetilde{enum} = \mathit{root}(\widetilde{\mathrm{ET}}, \widetilde{\mathrm{MOF}})$. The *isValueOf* relation between a literal $\widetilde{l}$ and its corresponding enumeration type ET is then formally defined by the equivalence

$$\boxed{\widetilde{l} : \mathrm{ET} \iff \widetilde{l} \in [\![\mathrm{ET}]\!]_{\mathrm{MOF}}}$$

For example, the algebraic semantics of the `RDataType` enumeration type is defined as follows:

$$[\![\mathrm{RDataType}]\!]_{\mathrm{MOF}} = \{\texttt{VARCHAR, NUMBER, BOOLEAN, DATE, DECIMAL}\}.$$

Primitive Types

The *defineParameter* function is defined for PRIMITIVETYPE instances as follows:

$$\mathit{defineParameter}(\widetilde{pt}, \widetilde{\mathcal{M}}) = (\emptyset,\ \emptyset,\ \emptyset,\ \emptyset),$$

where $\widetilde{pt}$: PRIMITIVETYPE and $\widetilde{\mathcal{M}}$: MOF.

Object Types

In the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory, the algebraic notion of *object type* is generically given by means of the sort $\mathit{Object\#\mathcal{M}}$. Terms of sort $\mathit{Object\#\mathcal{M}}$ are defined by means of the constructor

$$<\_:\_\,|\,\_> : \mathit{Cid\#\mathcal{M}}\ \ \mathit{Oid\#\mathcal{M}}\ \ \mathit{PropertySet\#\mathcal{M}} \to \mathit{Object\#\mathcal{M}},$$

which is provided by the theory $\texttt{EXT-MODEL}\{\mathcal{M}\}$ theory.

Defining the algebraic semantics of an object type $\widetilde{\mathrm{OT}}$ involves the definition of the object identifiers and the properties that may be involved in the definition of a specific object $\widetilde{o}$: OT in a model definition

$\widetilde{M} : \mathcal{M}$. Object type specialization relationships must be also taken into account. Therefore, we need to define the carrier of the sorts $\mathit{Oid\#\mathcal{M}}$, $\mathit{Cid\#\mathcal{M}}$ and $\mathit{PropertySet\#\mathcal{M}}$ for a specific object type definition $\widetilde{\mathrm{OT}}$. In addition, when the semantics of a property is defined, the following types may be involved: primitive types, enumeration types, object identifier types, and OCL collection types that are instantiated with any of the previous types. The algebraic representation of an object type definition $\widetilde{\mathrm{OT}}$ is defined as follows:

$$\begin{aligned}
&\mathit{defineParameter}(\widetilde{cl}, \widetilde{\mathcal{M}}) = \\
&\qquad \mathit{defineParameter}_{\mathit{Oid}}(\widetilde{cl}) \ \cup\ \mathit{defineParameter}_{\mathit{Cid}}(\widetilde{cl}) \ \cup \\
&\qquad \mathit{defineParameter}_{<_s}(\widetilde{cl}, \widetilde{\mathcal{M}}) \ \cup \\
&\qquad \bigcup_{\widetilde{p} \,\in\, \{\widetilde{p} :\, \textsc{Property} \,\mid\, \widetilde{p} \,\in\, \widetilde{cl}.\mathit{ownedAttribute}(\widetilde{\mathcal{M}})\}} \mathit{defineParameter}(\widetilde{p}, \widetilde{\mathcal{M}})
\end{aligned}$$

where: $\widetilde{cl} : \textsc{Class}$; $\widetilde{cl} = \mathit{root}(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}})$; $\widetilde{\mathcal{M}} : \mathrm{MOF}$; (1) the function

$$\mathit{defineParameter}_{\mathit{Oid}} : [\![\textsc{Class}]\!]_{\mathrm{MOF}} \longrightarrow \mathit{SpecMEL}$$

provides a MEL theory that represents identifier types for object types; (2) the function

$$\mathit{defineParameter}_{\mathit{Cid}} : [\![\textsc{Class}]\!]_{\mathrm{MOF}} \longrightarrow \mathit{SpecMEL}$$

provides a MEL theory that represents the set of names for object types; (3) the function

$$\mathit{defineParameter}_{\mathit{Prop}} : [\![\textsc{Property}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow \mathit{SpecMEL}$$

provides a MEL theory that permits defining properties in an object $\widetilde{o} : \mathrm{OT}$; and (4) the function

$$\mathit{defineParameter}_{<_s} : [\![\textsc{Class}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow \mathit{SpecMEL}$$

provides a MEL theory that represents object type specialization relationships as subsort relationships.

Consider, for example, the $\widetilde{\mathrm{RDBMS}}$ metamodel definition, where the TABLE object type, denoted by $\widetilde{\textsc{Table}}$, is specified in Maude notation as

```
< oid#Class('Table) : Class | name : "Table",
    isAbstract : false,
    ownedAttribute : OrderedSet{
        oid#Property('prop0) :: oid#Property('prop1) ::
        oid#Property('prop2) :: oid#Property('prop3)
    },
    superClass : OrderedSet{ oid#Class('RModelElement) }
    package : ...
>
< oid#Property('prop0) : Property |
    name : "schema", lower : 1, upper: 1,
    isOrdered, isUnique,
    isComposite = true,
    opposite = ...,
    type : oid#Class('Schema),
    class : oid#Class('Table)
>
< oid#Property('prop1) : Property |
    name : "column", lower : 0, upper: -1,
    isOrdered = true, isUnique = true,
    isComposite = false,
    opposite = ...,
    type : oid#Class('Column),
    class : oid#Class('Table)
>
< oid#Property('prop2) : Property |
    name : "key", lower : 0, upper: 1,
    isOrdered, isUnique,
    isComposite = false,
    opposite = ...,
    type : oid#Class('PrimaryKey),
    class : oid#Class('Table)
>
< oid#Property('prop3) : Property |
    name : "foreignKey", lower : 0, upper: -1,
    isOrdered = true, isUnique = true,
    isComposite = false,
    opposite = ...,
    type : oid#Class('PrimaryKey),
    class : oid#Class('Table)
>.
```

In subsequent paragraphs, we use this example to obtain the theory that defines this object type.

Algebraic Semantics of Object Types OT

The algebraic semantics of an object type is then given by the set of all the objects that can be defined either as instances of the object type, i.e., a class, or as instances of any of its subtypes. The algebraic semantics of an object type definition $\widetilde{\mathrm{OT}}$, such that $\widetilde{\mathrm{OT}} : \mathrm{MOF}$, $\mathit{root}(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}) : \textsc{Class}$, and $\mathit{tree}(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}) \subseteq \mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$, is defined as follows:

$$\boxed{\begin{aligned}[\![\mathrm{OT}]\!]_{\mathrm{MOF}} = \{\widetilde{o} \mid \quad & \widetilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\mathit{Object}\#\mathcal{M}} \wedge \\ & \mathit{class}(\widetilde{o}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\mathit{ClassSort}(\mathit{root}(\widetilde{\mathrm{OT}},\widetilde{\mathrm{MOF}}))}\}.\end{aligned}}$$

The *isValueOf* relation between an object $\widetilde{o}$ and an object type definition $\widetilde{\mathrm{OT}}$ is called *instanceOf*, and is defined by means of the equivalence

$$\boxed{\widetilde{o} : \mathrm{OT} \quad \Leftrightarrow \quad \widetilde{o} \in [\![\mathrm{OT}]\!]_{\mathrm{MOF}}.}$$

Object Type Names

In the $\mathit{defineParameter}(\widetilde{cl}, \widetilde{\mathcal{M}})$ theory, each Class instance $\widetilde{cl}$ is defined as a new sort and a constant, both of them with the name of the class. Recall the set of sort names *Sorts* in *SpecMEL*, and the set *Ops* of operators that can be declared in *SpecMEL* for a mel theory. To obtain the sort that corresponds to a Class instance $\widetilde{cl}$, i.e., $\widetilde{cl} : \textsc{Class}$, we define the function

$$\mathit{ClassSort} : [\![\textsc{Class}]\!]_{\mathrm{MOF}} \longrightarrow \mathit{Sorts},$$

which is defined by means of the equation

$$\mathit{ClassSort}(\widetilde{cl}) = \widetilde{cl}.\mathit{name}.$$

Similarly, the partial function

$$\mathit{ClassOp} : [\![\textsc{Class}]\!]_{\mathrm{MOF}} \rightsquigarrow \mathit{Ops}$$

obtains the declaration of the corresponding constant, when the Class instance is not abstract. In this case, the function is defined by means of the equation

$$\mathit{ClassOp}(\widetilde{cl}) = (\widetilde{cl}.\mathit{name} : \to \mathit{ClassSort}(\widetilde{cl})),$$

where $\widetilde{cl} : \textsc{Class}$ and $\widetilde{cl}.\mathit{abstract} = \mathit{false}$.

Abstract classes are defined as those that cannot be instantiated ([90]). The name of an abstract class C is *not* specified with a constant $C\ :\ C$, so that objects in a metamodel definition $\widetilde{\mathcal{M}}$ cannot have C as their type. Therefore, the function

$$\mathit{defineParameter}_{\mathit{Cid}} : [\![\textsc{Class}]\!]_{\mathrm{MOF}} \longrightarrow \mathit{SpecMEL}$$

is defined as follows:

$$\begin{aligned} \mathit{defineParameter}_{\mathit{Cid}}(\widetilde{cl}) = \;& (\{\mathit{ClassSort}(\widetilde{cl})\}, \emptyset, \{\mathit{ClassOp}(\widetilde{cl})\}, \emptyset) \\ & \mathit{when}\ \widetilde{cl}.\mathit{abstract} = \mathit{false} \end{aligned}$$

$$\begin{aligned} \mathit{defineParameter}_{\mathit{Cid}}(\widetilde{cl}) = \;& (\{\mathit{ClassSort}(\widetilde{cl})\}, \emptyset, \emptyset, \emptyset) \\ & \mathit{when}\ \widetilde{cl}.\mathit{abstract} = \mathit{true} \end{aligned}$$

In the example of the RDBMS metamodel, the theory $\mathit{defineParameter}_{\mathit{Cid}}(\mathit{root}(\widetilde{\text{TABLE}}, \widetilde{\text{MOF}}), \widetilde{\text{RDBMS}})$ is a theory with a single sort and a single constant, specified in Maude notation as follows,

```
sort Table .
op Table : -> Table .
```

Object Type Identifiers

In the MOF framework, each class instance has an associated identifier that distinguishes it from the others. An object identifier is obtained by means of the *oid* operator from an object. Object identifiers permit considering a model as a graph, where class instances are nodes and object-typed properties are edges between nodes. Identifier types are represented as sorts and constants for these sorts. An identifier sort is obtained from a CLASS instance $\widetilde{cl}$ by means of the partial operator

$$\mathit{OidSort} : [\![\text{CLASS}]\!]_{\text{MOF}} \rightsquigarrow \mathit{Sorts},$$

which is defined over CLASS instances $\widetilde{cl}$ by the equation

$$\mathit{OidSort}(\widetilde{cl}) = \widetilde{cl}.\mathit{name}.$$

Constructors for identifier values are generated from Class instances by means of the operator

$$\mathit{OidOp} : [\![\text{CLASS}]\!]_{\text{MOF}} \rightsquigarrow \mathit{Ops},$$

which obtains the declaration of the corresponding constant by the equation

$$\mathit{OidOp}(\widetilde{cl}) = (\mathit{oid}\#\widetilde{cl}.\mathit{name} :\ \mathit{Qid}\ \to \mathit{OidSort}(\widetilde{cl})),$$

where Qid is a sort for identifiers in Maude, $\widetilde{cl}$: CLASS and $\widetilde{cl}.\mathit{abstract} = \mathit{false}$.

Each object type has its own identifier type. The identifier type is related to the class type by means of a function $\mathit{oidType} : \mathit{Sorts} \to \mathit{Sorts}$ that maps a class sort to its corresponding identifier sort, i.e.,

$$\mathit{oidType} : \mathit{ClassSort}(\widetilde{cl}) \mapsto \mathit{OidSort}(\widetilde{cl}),$$

where $\widetilde{cl}$: CLASS. Object type specialization is algebraically represented by means of subsorts between the sorts of the corresponding class names, as discussed below. *oidType* is a monotonic function that preserves the partial order that is defined by this subsort relation, i.e., if $\widetilde{c1}$, $\widetilde{c2}$: CLASS, then:

$$\begin{aligned} & \mathit{ClassSort}(\widetilde{c1}) \ < \ \mathit{ClassSort}(\widetilde{c2}) \quad \Leftrightarrow \\ & \mathit{oidType}(\mathit{ClassSort}(\widetilde{c1})) \ < \ \mathit{oidType}(\mathit{ClassSort}(\widetilde{c2})). \end{aligned}$$

Therefore, the $\mathit{defineParameter}_{\mathit{Oid}} : [\![\text{CLASS}]\!]_{\text{MOF}} \to \mathit{SpecMEL}$ function is defined as follows:

$$\begin{aligned} \mathit{defineParameter}_{\mathit{Oid}}(\widetilde{cl}) = \;& (\{\mathit{OidSort}(\widetilde{cl})\}, \emptyset, \{\mathit{OidOp}(\widetilde{cl})\}, \emptyset) \\ & \mathit{when}\ \widetilde{cl}.\mathit{abstract} = \mathit{false} \end{aligned}$$

$$\begin{aligned} \mathit{defineParameter}_{\mathit{Oid}}(\widetilde{cl}) = \;& (\{\mathit{OidSort}(\widetilde{cl})\}, \emptyset, \emptyset, \emptyset) \\ & \mathit{when}\ \widetilde{cl}.\mathit{abstract} = \mathit{true} \end{aligned}$$

For the TABLE object type, its identifier type is represented as the sort `oid#Table`, the constructor `oid#Table :  Qid -> oid#Table`. An instance of the TABLE object type can then be defined as

```
< oid#Table('Foo) :  Table | ...  >.
```

Object Type Properties

An object type OT is defined with a collection of PROPERTY instances describing its properties. A PROPERTY instance $\widetilde{p}$ in a metamodel definition $\widetilde{\mathcal{M}}$: MOF is given by an object $\widetilde{p}$, such that $\widetilde{p}$: PROPERTY and $\widetilde{p} \in \widetilde{\mathcal{M}}$. As described in Section 6.2.1, a property definition is described by means of the following metaproperties: *name*, *ordered*, *unique*, *lower*, *upper*, *type* and *defaultValue*. The meta-properties *ordered*, *unique*, *lower* and *upper* constitute the multiplicity metadata of the property and permit, together with the *type* metaproperty, obtaining the algebraic type for the corresponding property constructor.

When an object $\widetilde{o}$: OT is created in a model definition $\widetilde{M}$ such that $\widetilde{M} : \mathcal{M}$, each property that is defined in the object type definition $\widetilde{\mathrm{OT}}$ can be initialized in $\widetilde{o}$, in which case the property is said to be *set*, or it can remain without any value, in which case the property is said to be *unset*. This is useful when a property is defined as required (meta-property *lower = 1*). Therefore, if we create an object $\widetilde{o}$ and the property is not still initialized, there is no error. Taking into account that properties can be set or unset, we algebraically represent them by means of the following operators:

- *Set properties:* $(prop : \_) \;\; : \; Type \rightarrow Property\#\mathcal{M}$, where *prop* is the name given to the property, and *Type* represents the type of the property, which can be a primitive type, an enumeration type, an object type, or an OCL collection type. *Property#*$\mathcal{M}$ is a sort that represents properties and that is a subsort of the *PropertySet#*$\mathcal{M}$ sort. For example, we define the property `name` of the class `RModelElement` of the RDBMS metamodel by means of the operator

  `(name:_) : String → Property#rdbms.`

  This operator can be used to define the property *name* of a column as follows:

  ```
  < oid#Column('Foo) : Column | name : "date", ... >
  ```

- *Unset properties:* are defined as constants *prop* : *Property#*$\mathcal{M}$, where *prop* is the name of the property. In the example, the operator `type : Property` allows the definition of the unset `type` property for the class `Column` in the RDBMS metamodel. We can add an unset `type` property to the `Class` instance as follows:

  ```
  < oid#Column('Foo) : Column | name : "date", type, ... >
  ```

A Property instance $\widetilde{p}$ is associated with a specific type $\widetilde{t}$ in the metamodel definition $\widetilde{\mathcal{M}}$, which is defined as an object $\widetilde{t}$: TYPE. Depending on the type $\widetilde{t}$ of a property, we can distinguish two kinds of properties:

- *Value-typed Properties or Attributes.* Properties of this kind are typed with DATATYPE instances. The above `name` property is an example of an attribute.

  If we consider the objects $\widetilde{o}$ that constitute a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, properties of this kind define the attributes of the nodes of the graph $graph(\widetilde{M}, \widetilde{\mathcal{M}})$.

- *Object-typed Properties or References.* Properties of this kind are typed with object types, so that the type definition that is referred to by means of the *type* meta-property is an object $\widetilde{t}$, such that $\widetilde{t}$: CLASS. Object-typed properties permit the definition of relationships between classes in a metamodel by using object identifiers as values. Object collections can then be viewed as graphs, where objects define graph nodes and object-typed properties define graph edges. For example, we can define a CLASS instance "Table" and a PROPERTY instance "name" that are related by means of their respective *ownedAttribute* and *class* properties:

  ```
  < oid#Class('class0) : Class |
      name : "Table",
      ownedAttribute : OrderedSet{oid#Property('prop0)} >
  < oid#Property('prop0) : Property |
      name : "name",
      class : oid#Class('class0) >
  ```

The *type* meta-property together with the multiplicity metadata, that is, the meta-properties *lower*, *upper*, *isOrdered* and *isUnique*, define a set of specific constraints on the acceptable values for the property type. These constraints are taken into account in the algebraic type that is assigned to the property by means of OCL collection types.

When the *upper* meta-property of a property definition $\widetilde{p}$ is > 1, an OCL collection type constitutes the type of the property in $reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$. When the *upper* meta-property of an object type property is 1, the type of the property is not represented as a collection type. In this case, we can distinguish two subcases:

Collection Type	Lower Bound	Upper Bound	isOrdered	isUnique
`[T]`	0	1	-	-
`Set{T}`	0	*	false	true
`OrderedSet{T}`	0	*	true	true
`Bag{T}`	0	*	false	false
`Sequence{T}`	0	*	true	false
`T`	1	1	-	-
`NeSet{T}`	1	*	false	true
`NeOrderedSet{T}`	1	*	true	true
`NeBag{T}`	1	*	false	false
`NeSequence{T}`	1	*	true	false

Table 6.1: Generic collection types instantiated with a sort `s`, depending on multiplicity metadata.

(i) when the lower bound is 0, indicating that the value of this property in an object can be a *null* value, and (ii) when the lower bound is 1, indicating that a property value is *required* for this property. In the former case, the type is represented by the kind of the corresponding algebraic sort, so that null values can be used. For example, the type of the property `key` of the `Table` class definition in the metamodel definition $\widetilde{\text{RDBMS}}$ is the kind $[Oid]$, so that the constant `nullOid` can be used to define a null value. In the latter case, the type is represented by the corresponding sort, so that null values are not allowed. For example, the type of the property `schema` of the `Table` class definition in the metamodel definition $\widetilde{\text{RDBMS}}$ is `Oid`. Therefore, if the `nullOid` constant is used as value for this property, it will be considered an error. Table 6.1 summarizes the combinations of multiplicity meta-property values of a specific property definition $\widetilde{p}$ that are used to obtain the corresponding type of the property.

Recall the set of sort names *Sorts* in *SpecMEL*. We define the function

$$sortName : [\![\textsc{Type}]\!]_{\text{MOF}} \longrightarrow Sorts$$

to obtain the sort that corresponds to the type of a property. This function is defined by the equation $sortName(\widetilde{t}) = \widetilde{t}.name$, where $\widetilde{t} : \textsc{Type}$. We define the domain *MELTypeExpression* by means of the equation $MELTypeExpression = Sorts \cup \{[s] \mid s \in Sorts\}$. The function

$$PropertyType : [\![\textsc{Property}]\!]_{\text{MOF}} \otimes [\![\text{MOF}]\!]_{\text{MOF}} \longrightarrow MELTypeExpression$$

obtains the type expression for the constructor of a specific property $\widetilde{p} : \textsc{Property}$ that is defined in a metamodel definition $\widetilde{\mathcal{M}}$. This function queries the multiplicity metadata of the Property instance $\widetilde{p}$ to obtain a suitable type for the property. The type of a property is always a Type instance $\widetilde{t}$, which can be a PrimitiveType instance, an Enumeration instance, or a Class instance. Given a Property instance $\widetilde{p}$, and a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{p} \in \widetilde{\mathcal{M}}$, the function *PropertyType* is defined by means of the following equalities[11]:

$$PropertyType(\widetilde{p}, \widetilde{\mathcal{M}}) = [sortName(\widetilde{p}.type(\widetilde{\mathcal{M}}))] \quad where\ \ \widetilde{p}.lower = 0 \wedge \widetilde{p}.upper = 1$$

$$PropertyType(\widetilde{p}, \widetilde{\mathcal{M}}) = Set\{sortName(\widetilde{p}.type(\widetilde{\mathcal{M}}))\}$$
$$where\ \ \widetilde{p}.lower = 0 \wedge \widetilde{p}.upper = -1 \wedge \widetilde{p}.ordered = false \wedge \widetilde{p}.unique = true$$

$$PropertyType(\widetilde{p}, \widetilde{\mathcal{M}}) = OrderedSet\{sortName(\widetilde{p}.type(\widetilde{\mathcal{M}}))\}$$
$$where\ \ \widetilde{p}.lower = 0 \wedge \widetilde{p}.upper = -1 \wedge \widetilde{p}.ordered = true \wedge \widetilde{p}.unique = true$$

$$PropertyType(\widetilde{p}, \widetilde{\mathcal{M}}) = Bag\{sortName(\widetilde{p}.type(\widetilde{\mathcal{M}}))\}$$
$$where\ \ \widetilde{p}.lower = 0 \wedge \widetilde{p}.upper = -1 \wedge \widetilde{p}.ordered = false \wedge \widetilde{p}.unique = false$$

$$PropertyType(\widetilde{p}, \widetilde{\mathcal{M}}) = Sequence\{sortName(\widetilde{p}.type(\widetilde{\mathcal{M}}))\}$$
$$where\ \ \widetilde{p}.lower = 0 \wedge \widetilde{p}.upper = -1 \wedge \widetilde{p}.ordered = true \wedge \widetilde{p}.unique = false$$

[11]We use the name of the sort that corresponds to a type, by means of the *sortName* function, as the view name that instantiates the `OCL-COLLECTION-TYPES{T :: TRIV}` theory, in order to obtain the corresponding OCL collection type.

$$PropertyType(\widetilde{p},\widetilde{\mathcal{M}}) = sortName(\widetilde{p}.type(\widetilde{\mathcal{M}})) \quad where \;\; \widetilde{p}.lower = 1 \wedge \widetilde{p}.upper = 1$$

$$PropertyType(\widetilde{p},\widetilde{\mathcal{M}}) = NeSet\{sortName(\widetilde{p}.type(\widetilde{\mathcal{M}}))\}$$
$$where \;\; \widetilde{p}.lower = 1 \wedge \widetilde{p}.upper = -1 \wedge \widetilde{p}.ordered = false \wedge \widetilde{p}.unique = true$$

$$PropertyType(\widetilde{p},\widetilde{\mathcal{M}}) = NeOrderedSet\{sortName(\widetilde{p}.type(\widetilde{\mathcal{M}}))\}$$
$$where \;\; \widetilde{p}.lower = 1 \wedge \widetilde{p}.upper = -1 \wedge \widetilde{p}.ordered = true \wedge \widetilde{p}.unique = true$$

$$PropertyType(\widetilde{p},\widetilde{\mathcal{M}}) = NeBag\{sortName(\widetilde{p}.type(\widetilde{\mathcal{M}}))\}$$
$$where \;\; \widetilde{p}.lower = 1 \wedge \widetilde{p}.upper = -1 \wedge \widetilde{p}.ordered = false \wedge \widetilde{p}.unique = false$$

$$PropertyType(\widetilde{p},\widetilde{\mathcal{M}}) = NeSequence\{sortName(\widetilde{p}.type(\widetilde{\mathcal{M}}))\}$$
$$where \;\; \widetilde{p}.lower = 1 \wedge \widetilde{p}.upper = -1 \wedge \widetilde{p}.ordered = true \wedge \widetilde{p}.unique = false$$

The function

$$defineParameter_{Prop} : [\![\textsc{Property}]\!]_{\text{MOF}} \otimes [\![\text{MOF}]\!]_{\text{MOF}} \longrightarrow SpecMEL$$

provides the operators that permit defining property values in a specific object. The function is defined by the equation

$$\begin{array}{ll} defineParameter_{Prop}(\widetilde{p},\widetilde{\mathcal{M}}) = & (\\ & \quad \{Property\#\mathcal{M}, PropertySet\#\mathcal{M}\}, \\ & \quad \{(Property\#\mathcal{M} < PropertySet\#\mathcal{M})\}, \\ & \quad \{(\widetilde{p}.name \;:\to\; Property\#\mathcal{M}), \\ & \quad (\widetilde{p}.name : \_ \;:\; PropertyType(\widetilde{p},\widetilde{\mathcal{M}}) \to\; Property\#\mathcal{M})\}, \\ & \quad \emptyset \\ &). \end{array}$$

In the example, the theory $reflect_{\text{MOF}}(\widetilde{\textsc{Table}}, \widetilde{\text{RDBMS}})$, which only has sorts and operators, is specified in Maude notation as follows:

```
sorts Table oid#Table .
op Table : -> Table .
op oid#Table : Qid -> oid#Table .
op schema : -> Property#rdbms .
op schema : Oid -> Property#rdbms .
op column : -> Property#rdbms .
op column : OrderedSet{Oid} -> Property#rdbms .
op key : -> Property#rdbms .
op key : [Oid] -> Property#rdbms .
op foreignKey : -> Property#rdbms .
op foreignKey : OrderedSet{Oid} -> Property#rdbms .
```

Object Type Specialization Relation $<_s$

A *specialization* is a taxonomic relationship between two object types. This relationship specializes a general object type into a more specific one. A specialization relation among object type definitions $\widetilde{\text{OT}}$ in a metamodel definition $\widetilde{\mathcal{M}}$: MOF is given by a set of specialization relationships between the class definitions that participate in their respective object type definition.

Given two object type definitions $\widetilde{\text{OT1}}$ and $\widetilde{\text{OT2}}$, such that

$$tree(\widetilde{\text{OT1}}, \widetilde{\text{MOF}}), tree(\widetilde{\text{OT2}}, \widetilde{\text{MOF}}) \subseteq tree(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}}),$$

we use the notation $\widetilde{\text{OT1}} <_s \widetilde{\text{OT2}}$ to indicate that the object type definition $\widetilde{\text{OT1}}$ specializes the object type definition $\widetilde{\text{OT2}}$. We define the specialization relationship $\widetilde{\text{OT1}} <_s \widetilde{\text{OT2}}$ by means of the equivalence

$$\begin{aligned}\widetilde{\mathrm{OT1}} <_s \widetilde{\mathrm{OT2}} \iff\ & \exists \widetilde{cl1}, \widetilde{cl2}(\widetilde{cl1}, \widetilde{cl2} : \textsc{Class} \wedge \\ & \widetilde{cl1} = root(\widetilde{\mathrm{OT1}}, \widetilde{\mathrm{MOF}}) \wedge \\ & \widetilde{cl2} = root(\widetilde{\mathrm{OT2}}, \widetilde{\mathrm{MOF}}) \wedge \\ & \widetilde{cl2} = \widetilde{cl1}.superClass(\widetilde{\mathcal{M}})).\end{aligned}$$

In the metamodel definition $\widetilde{\mathrm{RDBMS}}$, we define the Class instance

```
< OID2 : Class | name : "Table", superClass : OrderedSet{ OID1 }, PS2 >,
```

specializes the Class instance

```
< OID1 : Class | name : "RModelElement", PS1 >,
```

by means of the *superClass* property value.

Each specialization relationship in $\widetilde{\mathcal{M}}$ is mapped to a subsort relationship between the corresponding class sorts, i.e.,

$$\begin{aligned}& defineParameter_{<_s}(\widetilde{cl1}, \widetilde{\mathcal{M}}) = \\ & \quad (\emptyset, \\ & \quad \{ClassSort(\widetilde{cl1}) < ClassSort(\widetilde{cl2}) \mid \\ & \qquad \widetilde{cl2} : \textsc{Class} \wedge \widetilde{cl2} \in \widetilde{cl1}.superClass(\widetilde{\mathcal{M}})\} \cup \\ & \quad \{OidSort(\widetilde{cl1}) < OidSort(\widetilde{cl2}) \mid \\ & \qquad \widetilde{cl2} : \textsc{Class} \wedge \widetilde{cl2} \in \widetilde{cl1}.superClass(\widetilde{\mathcal{M}})\}, \\ & \quad \emptyset, \\ & \quad \emptyset) \\ & \ when\ \widetilde{cl1} : \textsc{Class} \wedge \widetilde{cl1}.superClass(\widetilde{\mathcal{M}}) \neq \emptyset\end{aligned}$$

$$\begin{aligned}& defineParameter_{<_s}(\widetilde{cl1}, \widetilde{\mathcal{M}}) = \\ & \quad (\emptyset, \{ClassSort(\widetilde{cl1}) < Cid\#\mathcal{M}, OidSort(\widetilde{cl1}) < Oid\#\mathcal{M}\}, \emptyset, \emptyset) \\ & \ when\ \widetilde{cl1} : \textsc{Class} \wedge \widetilde{cl1}.superClass(\widetilde{\mathcal{M}}) = \emptyset\end{aligned}$$

In the RDBMS example, we algebraically define the specialization relationship between the object types $\widetilde{RModelElement}$ and $\widetilde{Table}$ as the subsorts `Table < RModelElement` and `oid#Table < oid#RModelElement`. The supersorts of the resulting subsort hierarchy are defined as subsorts of the `Cid#rdbms` and `Oid#rdbms` sorts, for object type name sorts and object identifier sorts, respectively. In this way, we can define a table instance as `< oid#Table('Foo) : Table | name : "date", ...>`, where the `name` property is defined for the RModelElement object type.

Algebraic Semantics of the Specialization Relation $<_s$

A specialization relationship OT1 $<_s$ OT2, between two object types of the metamodel $\mathcal{M}$ is defined as a subset inclusion between the carriers of the corresponding name sorts in the initial algebra of the $reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory, i.e., we have the implication

$$\boxed{[\![\mathrm{OT1} <_s \mathrm{OT2}]\!]_{\mathrm{MOF}} \implies [\![\mathrm{OT1}]\!]_{\mathrm{MOF}} \subseteq [\![\mathrm{OT2}]\!]_{\mathrm{MOF}}.}$$

6.3.3 Name Strategy

In this section, the object types that are defined in the metamodel definition $\widetilde{\mathrm{MOF}}$ are subtypes of the NamedElement object type, and, thereby, they contain a property *name*. This property is used to define sort names, operator names and view names in the $reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory. However, the same name may be used for different elements within the same metamodel definition $\widetilde{\mathcal{M}}$: two object types that are not related by means of a chain of specialization relationships may contain properties with the same name, two enumeration types can be defined with the same name in different packages, two object types can be defined with the same name in different packages, etc. These situations may lead to several problems. For example, different theories may have the same name, in the case of enumeration types with the same name. Another example is that the set of subsorts that is defined in the $defineParameter(root(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}), \widetilde{\mathcal{M}})$ theory may not represent the specialization relation $<_s$. Assume that a metamodel definition $\widetilde{\mathcal{M}}$ is constituted by a package A, which contains a subpackage B. In the package A, the object types A1 and A2 are defined so that A2 $<_s$ A1. In the package B, the object types A1 and B are defined so that B $<_s$ A1. The resulting $\mathit{defineParameter}(\mathit{root}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}), \widetilde{\mathcal{M}})$ theory is, in Maude notation, as follows:

```
mod mod#𝓜 is
  sorts A1 A2 B .
  subsorts A2 B < A1 .
  op A1 :  -> A1 .  op A2 :  -> A2 .  op B : -> B .
  ...
endm
```

Therefore, the object type A1 of the A and B packages is considered to be the same. To solve this problem, we use a strategy to structure the names of the objects that consitute $\widetilde{\mathcal{M}}$ by taking into account the containment relation that is defined for a metamodel definition $\widetilde{\mathcal{M}}$ as $<_c$ $(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$. This strategy is defined by means of the function

$$\mathit{buildName} : [\![\mathrm{NamedElement}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow \mathit{String}$$

as follows:

$$\mathit{buildName}(\widetilde{o}, \widetilde{\mathcal{M}}) = \widetilde{o}.\mathit{name} \quad \text{when } \widetilde{o} = \mathit{root}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$$
$$\mathit{buildName}(\widetilde{o}, \widetilde{\mathcal{M}}) = \mathit{buildName}(\mathit{container}(\widetilde{o}, \widetilde{\mathcal{M}}), \widetilde{\mathcal{M}}) + \text{"/"} + \widetilde{o}.\mathit{name} \quad \text{otherwise}$$

where the function

$$\mathit{container} : [\![\mathrm{NamedElement}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \rightsquigarrow [\![\mathrm{NamedElement}]\!]_{\mathrm{MOF}}$$

obtains the parent node of the node $\widetilde{o}$ in the tree $\mathit{tree}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$, when it exists, and $\widetilde{\mathrm{MOF}}$ is the MOF meta-metamodel definition. Note that $\widetilde{\mathrm{MOF}}$ is defined as a constant in the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory, so that it can always be used without any need of passing it as argument to the *buildName* function.

In the *reflect* function, and also in the $\mathit{reflect}_{MOF}$ function, whenever the name of a MOF object type instance is used, the *buildName* function is applied, although we omit this fact in their definition for the sake of simplicity. Taking into account the strategy of structured names, the theory $\mathit{defineParameter}(\mathit{root}(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}), \widetilde{\mathcal{M}})$ that corresponds to the the previous metamodel definition $\widetilde{\mathcal{M}}$ is specified, in Maude notation, as follows:

```
mod mod#𝓜 is
  sorts A/A1 A/A2 A/B/A1 A/B/B .
  subsort A/A2 < A/A1 .
  subsort A/B/B < A/B/A1 .
  op A/A1 :  -> A/A1 .  op A/A2 :  -> A/A2 .
  op A/B/A1 :  -> A/B/A1 .  op A/B/B : -> A/B/B .
  ...
endm
```

6.4 Reflecting the Algebraic Semantics: the Reflect Operator.

The logical reflective features of MEL, together with its logical framework capabilities, make it possible to *internalize* the representation $\Phi : \mathit{Spec}\mathcal{L} \longrightarrow \mathit{SpecMEL}$ of a formalism $\mathcal{L}$ in MEL, as an equationally-defined function $\overline{\Phi} : \mathit{Module}_{\mathcal{L}} \rightarrow \mathit{Module}$, where $\mathit{Module}_{\mathcal{L}}$ is an equationally defined data type representing specifications in $\mathcal{L}$, and *Module* is the data type whose terms, of the form $\overline{(\Sigma, E)}$, metarepresent MEL specifications of the form (Σ, E). We can apply this general method to the case of our algebraic semantics

$$reflect : SpecMOF \rightsquigarrow SpecMEL.$$

Specifically, we define the function

$$\mathit{reflect}_{\mathrm{MOF}} : \mathrm{MOF} \rightsquigarrow SpecMEL$$

so that $\mathit{reflect}(\widetilde{\mathcal{M}}, \varnothing) = \mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$. Then, the reflective internalization of the MOF algebraic semantics $\mathit{reflect}_{\mathrm{MOF}}$ becomes an equationally-defined function

$$\overline{\mathit{reflect}_{\mathrm{MOF}}} : \mathit{Configuration}\{\mathrm{MOF}\} \rightsquigarrow \mathit{Module}.$$

where *Module* is the sort whose terms represent MEL theories in the universal MEL theory (see [75]). This is a very powerful construction, which we have implemented in our Maude executable specification of the MOF algebraic semantics. It is very powerful because it makes available the algebraic semantics $\mathit{reflect}_{\mathrm{MOF}}$ itself as a computable function $\overline{\mathit{reflect}_{\mathrm{MOF}}}$, which we can use for many formal transformational purposes. For example, suppose that we want, given the data representation $\widetilde{\mathrm{MOF}}$ of the MOF metamodel, to compute the MEL theory that is its mathematical semantics. This MEL theory is precisely the one metarepresented in MEL as the term $\overline{\mathit{reflect}_{\mathrm{MOF}}}(\widetilde{\mathrm{MOF}})$ of sort *Module*.

The Eclipse Modeling Framework is an informal implementation of the MOF framework, where the meta-metamodel definition $\widetilde{\mathrm{MOF}}$ is substituted by the meta-metamodel definition $\widetilde{\mathrm{ECORE}}$. The equivalence of both metamodels is studied in [91]. Since our formalization can be applied to any MOF-like metamodel, we have formalized the $\widetilde{\mathrm{ECORE}}$ meta-metamodel as a MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\mathrm{ECORE})$, as shown in Appendix A. The metamodel definition $\widetilde{\mathrm{ECORE}}$, such that $\widetilde{\mathrm{ECORE}} : \mathrm{ECORE}$, is provided in [92].

As another example, the metamodel in Fig. 2.2 provides the concepts for modeling relational schema elements: SCHEMA, TABLE, COLUMN, FOREIGNKEY and PRIMARYKEY. RMODELELEMENT is an abstract object type that defines an attribute *name* that will be inherited by the rest of object types of the metamodel. In the metamodel there is one PRIMITIVETYPE instance ($\widetilde{\mathrm{STRING}}$) and one ENUMERATION instance $\widetilde{\mathrm{RDATATYPE}}$. Both of them define the data types that can be used to indicate the type of a COLUMN instance. To define a foreign key, a FOREIGNKEY instance must refer to one or several COLUMN instances of its containing TABLE instance and to one PRIMARYKEY instance. The RDBMS model type is provided as a metamodel definition $\widetilde{\mathrm{RDBMS}}$, such that $\widetilde{\mathrm{RDBMS}} : \mathrm{MOF}$, in Appendix B. The resulting theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{RDBMS}})$ is provided in Appendix C. Finally, the relational schema $\widetilde{\mathrm{rsPERSON}}$ such that $\widetilde{\mathrm{rsPERSON}} : \mathrm{RDBMS}$, which is shown at level M1 of the MOF framework in Fig. 2.2, is provided as a collection of objects in Appendix D.

6.5 Reifying the Algebraic Semantics: the Inverse Step.

Reifying a MOF metamodel that is used at the base sublevel of the level M2, in a MOF framework, constitutes an important feature of the MOF reflection. It permits the evolution of the formal semantics of a MOF metamodel, providing complete formal support for reflection. In our approach the reification of EMOF metamodels is defined by means of the *reify* function. The partial function

$$\mathit{reify} : SpecMEL \rightsquigarrow [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$$

maps a MEL theory that has been previously generated from a metamodel definition $\widetilde{\mathcal{M}}$ by means of the function *reflect*, i.e., the *reify* function satisfies the equation

$$\mathit{reify}(\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})) = \widetilde{\mathcal{M}}.$$

The algebraic semantics of a specific metamodel $\widetilde{\mathcal{M}}$ can be metarepresented as data by using the MEL reflective features. Then, *reify* is an equationally-defined function whose domain and co-domain sorts are: $\overline{\mathit{reify}} : \mathit{Module} \rightarrow \mathit{Configuration}\{\mathrm{MOF}\}$.

6.6 Summary

In this section, we have provided the algebraic semantics of a MOF metamodel definition $\widetilde{\mathcal{M}}$ by means of a MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$. In particular, the algebraic semantics of the following notions has been provided:

- *model type* and *structural conformance relation*;
- *primitive type*, *enumeration type*, *OCL collection types*, and *isValueOf* relation for each one of these types;
- *object type* and *instanceOf* relation;
- *specialization relation* $<_s$; and
- *composition relation* $<_c$.

Taking into account an arbitrary model type $\mathcal{M}$, such that $\widetilde{\mathcal{M}}$: MOF, a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, can be considered as:

- a *graph* given by the pair (V, E), where V is the set of nodes given by $\widetilde{M}$, and E is the set of edges given by the set of object-typed properties between pairs of objects in $\widetilde{M}$; and
- as a *forest* given by the pair $(\widetilde{M}, <_c)$.

In addition, the specialization relation $<_s$ that is defined for $\mathcal{M}$ permits classifying the objects that constitute $\widetilde{M}$.

A metamodel specification $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ is constituted by a metamodel definition $\widetilde{\mathcal{M}}$ and a set $\widetilde{\mathcal{C}}$ of OCL constraint definitions. The algebraic semantics of $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ is given as a MEL theory *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, which is defined by composing *reflect*$_{\text{MOF}}(\widetilde{\mathcal{M}})$ and the algebraic semantics of the OCL language.

In Section 7, the types that are provided by the *reflect*$_{\text{MOF}}$ function are extended by adding the OCL operators for each one of these types, and the algebraic semantics for OCL expressions is defined by means of a mapping

$$\mathit{reflect} : (\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}) \quad \mapsto \quad (S, \leqslant, \Omega, E \cup A),$$

which formally defines the *constrained conformance relation* $\widetilde{M} : (\mathcal{M}, \mathcal{C})$.

Chapter 7

Algebraic Constrained Conformance Relation

A metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ is constituted by a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF, and a set $\widetilde{\mathcal{C}}$ of OCL constraints $\widetilde{c}$ for $\widetilde{\mathcal{M}}$. The abstract syntax of the OCL language is provided as a MOF 2.0 metamodel definition $\widetilde{\text{OCL}}$, such that $\widetilde{\text{OCL}}$: MOF, in the OCL standard specification (see [24], Chapter 8). The types that are defined in $\widetilde{\text{OCL}}$ are algebraically defined in the theory $\mathit{reflect}_{MOF}(\widetilde{\text{OCL}})$, providing the $[\![\text{OCL}]\!]_{\text{MOF}}$ model type as the set of well-formed OCL expressions. OCL constraints $\widetilde{c}$ can then be defined as model definitions $\widetilde{c}$: OCL by taking into account a specific metamodel definition $\widetilde{\mathcal{M}}$. An OCL constraint $\widetilde{c}$ is constituted by a context definition that refers to a type definition in $\widetilde{\mathcal{M}}$, denoted by $\mathit{context}(\widetilde{c})$, and by a body in the form of an OCL expression. Given a pair of the form $(\widetilde{\mathcal{M}}, \widetilde{c})$, $\widetilde{c}$ constitutes a *meaningful constraint* for $\widetilde{\mathcal{M}}$ iff $\mathit{context}(\widetilde{c}) \in \widetilde{\mathcal{M}}$[1].

We call *SpecMOF* to the set of metamodel specifications $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ that satisfy the condition

$$\forall \widetilde{c} : \text{OCL}\ (\widetilde{c} \in \widetilde{\mathcal{C}} \implies \mathit{context}(\widetilde{c}) \in \widetilde{\mathcal{M}}).$$

Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the partial function

$$\mathit{reflect} : \mathit{SpecMOF} \rightsquigarrow \mathit{SpecMEL}$$

is defined in this section. This function maps a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ to a MEL theory, providing the semantics for the following notions: (i) constrained model type $[\![(\mathcal{M}, \mathcal{C})]\!]_{\text{MOF}}$, (ii) OCL constraint satisfaction relation $\widetilde{M} \models \mathcal{C}$ for $\widetilde{M} : \mathcal{M}$, (iii) constrained conformance relation $\widetilde{M} : (\mathcal{M}, \mathcal{C})$ for $\widetilde{M} : \mathcal{M}$, and (iv) metamodel specification realization $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. When the *reflect* function is applied to a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory includes the $\mathit{reflect}_{\text{MOF}}(\widetilde{\mathcal{M}})$ theory, i.e.,

$$\mathit{reflect}_{\text{MOF}}(\widetilde{\mathcal{M}}) \subseteq \mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}),$$

where the $[\![\mathcal{M}]\!]_{\text{MOF}}$ model type, which is defined in the $\mathit{reflect}_{\text{MOF}}(\widetilde{\mathcal{M}})$ theory, is preserved in the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory. Recall the sort $\mathit{ConsistentModelType}\{\mathcal{M}\}$, also denoted by $(\mathcal{M}, \mathcal{C})$, that remains undefined in the $\mathit{reflect}_{\text{MOF}}(\widetilde{\mathcal{M}})$ theory. The $(\mathcal{M}, \mathcal{C})$ sort is a subsort of the $\mathcal{M}$ sort, and its semantics $[\![(\mathcal{M}, \mathcal{C})]\!]_{\text{MOF}}$ constrains the semantics $[\![\mathcal{M}]\!]_{\text{MOF}}$ of the $\mathcal{M}$ model type sort by taking into account the set $\mathcal{C}$ of OCL constraints. Therefore, we obtain the implication $\widetilde{M} : (\mathcal{M}, \mathcal{C}) \implies \widetilde{M} : \mathcal{M}$.

$[\![(\mathcal{M}, \mathcal{C})]\!]_{\text{MOF}}$ constitutes the *constrained model type* that is defined as data in the metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. The *reflect* function defines the constrained model type $[\![(\mathcal{M}, \mathcal{C})]\!]_{\text{MOF}}$, in the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory, by means of a membership axiom of the form

$$\widetilde{M} : \mathcal{M}\ \wedge\ \mathit{condition}_1(\widetilde{M}) = \mathit{true}\ \wedge \cdots \wedge\ \mathit{condition}_n(\widetilde{M}) = \mathit{true} \Longrightarrow \widetilde{M} : (\mathcal{M}, \mathcal{C}),$$

where each constraint definition $\widetilde{c}_i$, in $\widetilde{\mathcal{C}}$, corresponds to a boolean function $\mathit{condition}_i$ that is evaluated over a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, i.e., $\mathit{condition}_i(\widetilde{M})$. Therefore, when a model definition $\widetilde{M}$ satisfies

[1]The expression $\mathit{context}(\widetilde{c})$ denotes an object $\widetilde{o}$, such that $\widetilde{o} \in T_{\mathit{reflect}_{\text{MOF}}(\widetilde{\text{MOF}}), \mathit{Object\#MOF}}$ and $\widetilde{o} \in \widetilde{\mathcal{M}}$, as defined below.

all the constraints that are defined in $\widetilde{\mathcal{C}}$, $\widetilde{M}$ is considered a value of the constrained model type $(\mathcal{M},\mathcal{C})$. When a metamodel specification definition $(\widetilde{\mathcal{M}},\widetilde{\mathcal{C}})$ is given, whenever a model definition $\widetilde{M}$ of sort $(\mathcal{M},\mathcal{C})$ is defined in the $\mathit{reflect}(\widetilde{\mathcal{M}},\widetilde{\mathcal{C}})$ theory, we can assume that $\widetilde{M}$ satisfies the set $\widetilde{\mathcal{C}}$ of OCL constraints by definition. There is a subtle difference with other current approaches for the validation of OCL constraints: in our approach, OCL constraints do not have to be checked explicitly. They are instead taken into account in the semantics of the $(\mathcal{M},\mathcal{C})$ sort and are checked implicitly by evaluating the above membership axiom, so that a model definition $\widetilde{M}$ will satisfy the constraints $\widetilde{\mathcal{C}}$ iff its canonical form has sort $(\mathcal{M},\mathcal{C})$.

In subsequent sections, we provide: (1) the domain of the *reflect* function by defining the algebraic semantics of the metamodel definition $\widetilde{\mathrm{OCL}}$; (2) the operators of the types that are defined in a metamodel definition $\widetilde{\mathcal{M}}$, which may be used to define the conditions of the membership that specifies the OCL constraint satisfaction relation for a metamodel specification $(\mathcal{M},\mathcal{C})$; (3) the mappings of the *reflect* function that permit defining the aforementioned membership in the $\mathit{reflect}(\widetilde{\mathcal{M}},\widetilde{\mathcal{C}})$ theory; and (4) a formal definition for the following notions: (i) constrained model type $[\![(\mathcal{M},\mathcal{C})]\!]_{\mathrm{MOF}}$, (ii) OCL constraint satisfaction relation $\widetilde{M} \models \mathcal{C}$, (iii) constrained conformance relation $\widetilde{M} : (\mathcal{M},\mathcal{C})$, and (iv) metamodel specification realization $\mathit{reflect}(\widetilde{\mathcal{M}},\widetilde{\mathcal{C}})$.

7.1 Algebraic Semantics of the OCL Metamodel

The concrete syntax of the OCL language is provided as an attributed EBNF grammar in the OCL standard specification (see [24], Chapter 9). The abstract syntax of the OCL language is provided as a MOF 2.0 metamodel definition $\widetilde{\mathrm{OCL}}$, such that $\widetilde{\mathrm{OCL}}$: MOF, in the OCL standard specification (see [24], Chapter 8). Part of this metamodel is the OCL standard library, which provides the predefined types of the the OCL language and their operations (see [24], Chapter 11). The OCL standard library can be viewed as a metamodel definition $\widetilde{\mathrm{OclStdLib}}$, such that $\widetilde{\mathrm{OclStdLib}}$: MOF.

The metamodel definition $\widetilde{\mathrm{OCL}}$ is algebraically represented by the theory $\mathit{reflect}_{MOF}(\widetilde{\mathrm{OCL}})$. However, the $\mathit{reflect}_{MOF}(\widetilde{\mathrm{OCL}})$ theory only provides the types that are needed to define OCL expressions as model definitions $\widetilde{M}$, such that $\widetilde{M}$: OCL. To provide the algebraic semantics of the OCL language, the semantics of the predefined OCL type operators, which are defined in $\widetilde{\mathrm{OclStdLib}}$, also has to be provided.

In this section, we present the metamodel definition $\widetilde{\mathrm{OCL}}$, which is used to define the domain of the *reflect* function. The metamodel definition $\widetilde{\mathrm{OCL}}$ imports the metamodel definition $\widetilde{\mathrm{MOF}}$. This relationship is considered as a subcollection inclusion $\widetilde{\mathrm{MOF}} \subseteq \widetilde{\mathrm{OCL}}$, so that the object types that are defined in $\widetilde{\mathrm{MOF}}$ can also be used in $\widetilde{\mathrm{OCL}}$[2]. We denote by $\widetilde{\mathrm{OCL}}_{\vartriangle}$ the collection of objects that constitute the metamodel definition that is described in the OCL standard specification, i.e., $\widetilde{\mathrm{OCL}}_{\vartriangle} = \widetilde{\mathrm{OCL}} - \widetilde{\mathrm{MOF}}$.

Recall the function $\mathit{reflect}_{\mathrm{MOF}} : \mathrm{MOF} \longrightarrow \mathit{SpecMEL}$ that maps a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF, to a MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$, which defines the algebraic semantics of: (i) the types T that are defined in $\widetilde{\mathcal{M}}$, including a model type, primitive types, enumeration types, and object types; (ii) their corresponding *isValueOf* relation for values $\widetilde{v}$, i.e., $\widetilde{v}$: T; (iii) the specialization relation $<_s$ between object types in $\mathcal{M}$; and (iv) the containment relation $<_c$ between objects in a model $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$. Since $\widetilde{\mathrm{OCL}}_{\vartriangle}, \widetilde{\mathrm{OCL}}, \widetilde{\mathrm{MOF}}$: MOF, the algebraic semantics of the types that are defined in $\widetilde{\mathrm{OCL}}$ is provided by the equation

$$\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{OCL}}) = \mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}) \cup \mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{OCL}}_{\vartriangle}),$$

where the theories $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ and $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{OCL}}_{\vartriangle})$ are provided by means of the $\mathit{reflect}_{\mathrm{MOF}}$ function, detailed in Section 6.3.

The partial function

$$\mathit{reflect} : \mathit{SpecMOF} \rightsquigarrow \mathit{SpecMEL}$$

receives a metamodel specification of the form $(\widetilde{\mathcal{M}},\widetilde{\mathcal{C}})$ as argument, where $\widetilde{\mathcal{M}}$ is a metamodel definition such that $\widetilde{\mathcal{M}}$: MOF, and $\widetilde{\mathcal{C}}$ is a finite set of constraints $\widetilde{c}$, such that $\widetilde{c}$: OCL, that represent meaningful constraints for $\widetilde{\mathcal{M}}$.

The mappings between the concrete syntax and the abstract syntax of the OCL language are defined in the OCL specification. In our approach, we assume that these mappings are provided by a third-party tool.

[2]In the OCL specification, the metamodel definition $\widetilde{\mathrm{UML}}$ is taken into account instead of $\widetilde{\mathrm{MOF}}$, i.e., $\widetilde{\mathrm{UML}} \subseteq \widetilde{\mathrm{OCL}}$. However, we consider that $\widetilde{\mathrm{MOF}} \subseteq \widetilde{\mathrm{OCL}}$ since both $\widetilde{\mathrm{MOF}}$ and $\widetilde{\mathrm{UML}}$ metamodel definitions share a basic infrastructure of object types and primitive types, as stated in [22].

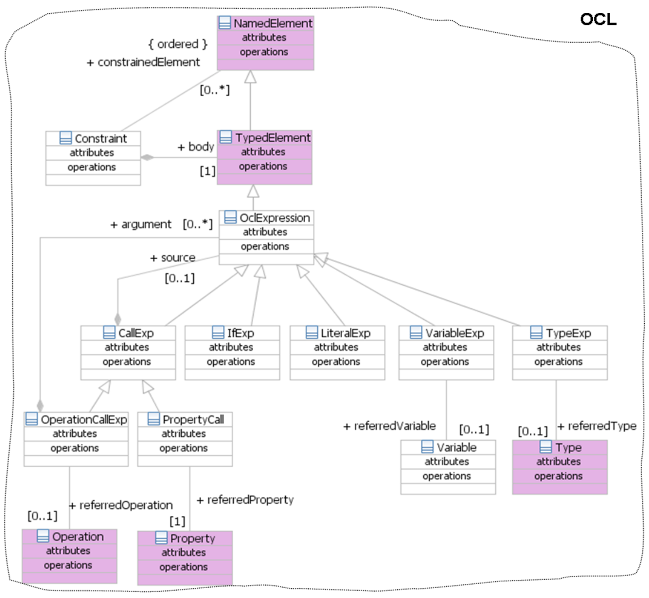


Figure 7.1: Expressions package of the OCL metamodel: core part.

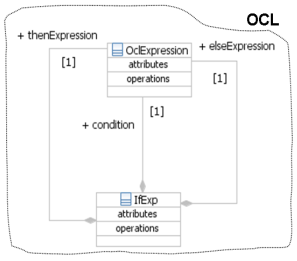


Figure 7.2: Expressions package of the OCL metamodel: ifThen expressions.

Therefore, whenever an OCL constraint is given in textual format by using the concrete syntax of the OCL language, we assume that it is provided as a model definition $\widetilde{c}$, such that $\widetilde{c}$: OCL.

In the subsequent section, we provide a brief introduction of the abstract syntax of the OCL language, i.e., the metamodel definition $\widetilde{\mathrm{OCL}_\Delta}$, by giving its graphical representation by means of class diagrams. As we have already mentioned in Section 6.2, this graphical representation is isomorphic to the definition of the model type OCL_Δ as a collection of objects $\widetilde{\mathrm{OCL}_\Delta}$: MOF, and is more readable as well.

7.1.1 Abstract Syntax of the OCL Language

In this section, we provide an overview of the metamodel definition $\widetilde{\mathrm{OCL}_\Delta}$. We focus on the *Expressions* package of the OCL metamodel, which defines the structure of the OCL expressions that can be used to define an OCL constraint. Our goal in this section consists in providing an enumeration of the object types that constitute $\widetilde{\mathrm{OCL}_\Delta}$. We refer to [31] for a better understanding of the OCL language, and, to the OCL specification [24] for a more detailed presentation of the metamodel definition $\widetilde{\mathrm{OCL}}$. We show the package expression of the metamodel definition $\widetilde{\mathrm{OCL}}$, in UML notation, in Fig. 7.1, 7.2, 7.3, 7.4, and 7.5. In this figures, the object types that are depicted with pink background belong to the metamodel definition $\widetilde{\mathrm{MOF}}$.

A Constraint instance $\widetilde{ct}$ represents an OCL constraint that is related to an object type of a metamodel definition $\widetilde{\mathcal{M}}$ by means of the *constrainedElement* property. The referred object type constitutes the *contextual type* of the constraint, and is represented as $\widetilde{ct}.constrainedElement(\widetilde{\mathcal{M}})$. Taking into account the containment relation $<_c$ that is defined in $\widetilde{\mathrm{OCL}_\Delta}$, a constraint definition $\widetilde{c}$, such that $\widetilde{c}$: OCL, can be viewed as a model tree $tree(\widetilde{c}, \widetilde{\mathrm{OCL}})$, where its root object is the Constraint instance $\widetilde{ct}$, i.e.,

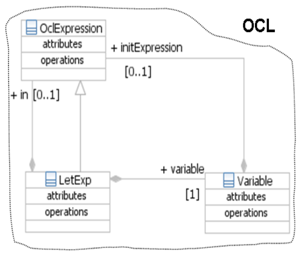


Figure 7.3: Expressions package of the OCL metamodel: let expressions.

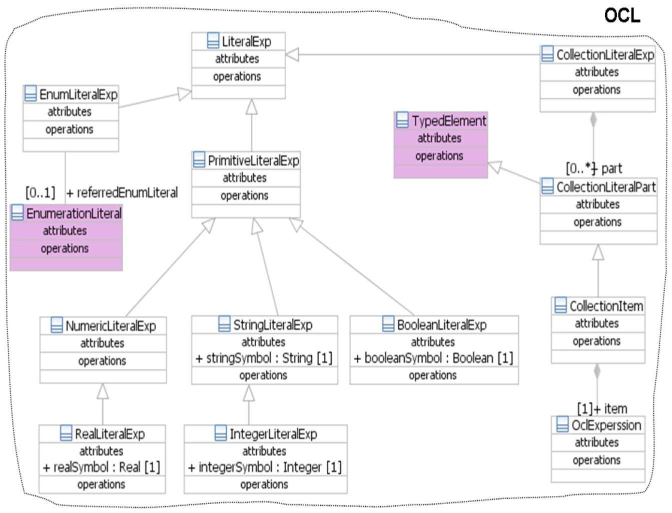


Figure 7.4: Expressions package of the OCL metamodel: literal expressions.

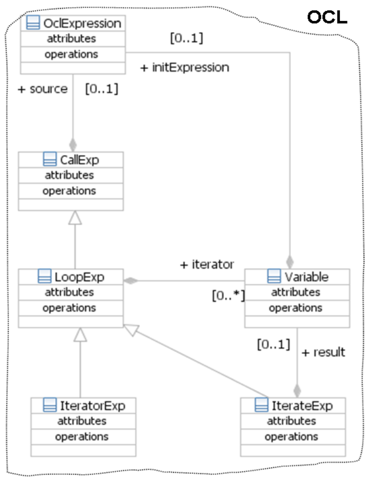


Figure 7.5: Expressions package of the OCL metamodel: loop expressions.

$root(\widetilde{c}, \widetilde{\mathrm{OCL}}) = \widetilde{ct}$. Given a specific metamodel definition $\widetilde{\mathcal{M}}$ and an OCL constraint definition $\widetilde{c}$, we say that $\widetilde{c}$ is a *meaningful OCL constraint* for $\widetilde{\mathcal{M}}$ if its contextual type is a type definition in the metamodel $\widetilde{\mathcal{M}}$, i.e., $\widetilde{ct}.constrainedElement(\widetilde{\mathcal{M}}) \neq \emptyset$.

OCL constraints are always evaluated for a single object $\widetilde{o}$, which is always an instance of the corresponding contextual type. In this case, the object $\widetilde{o}$ is called the *contextual instance*. In an OCL constraint definition, the contextual instance can be explicitly referred to by means of the *self* keyword.

An OclExpression instance represents an expression that can be evaluated in a given environment. OclExpression is the abstract super type of all other object types in the metamodel definition $\widetilde{\mathrm{OCL}_{\Delta}}$. Every OclExpression instance has a type that can be statically determined by analyzing the expression and its context. Evaluation of an OCL expression results in a value. Expressions with boolean result can be used as constraints (e.g., to specify an invariant of an object type).

The environment of an OclExpression instance defines what model elements are visible and can be referred to in an expression. Taking into account the tree structure of an OCL constraint definition $\widetilde{c}$ that is given by $tree(\widetilde{c}, \widetilde{\mathrm{OCL}})$, at the topmost level of $tree(\widetilde{c}, \widetilde{\mathrm{OCL}})$ the environment contains the *self* variable that refers to the contextual instance. On a lower level in $tree(\widetilde{c}, \widetilde{\mathrm{OCL}})$, the following variables can be introduced into the environment: *iterator* variables that are declared in a LoopExp instance, the *result* variable that can be defined in an IterateExp instance, and the variable that can be defined in a LetExp instance. All these object types are defined, among others, in the metamodel definition $\widetilde{\mathrm{OCL}}$ as follows:

- IfExp : An IfExp instance represents an OCL expression that results in two alternative expressions, *thenExpression* and *elseExpression*, depending on the evaluated value of a *condition*.
- LetExp: A LetExp instance represents a special expression that defines a new *variable* with an initial value. A variable defined by a LetExp instance cannot change its value. Its value corresponds to the evaluation of the initial expression *initExpression*. The variable is visible in the *in* expression.
- LoopExp: A LoopExp instance is an expression that represents an iteration construct over a *source* collection. It has an *iterator* variable that represents the elements of the source collection during the iteration process. The *body* expression is evaluated for each element in the collection. The result of a loop expression depends on the specific kind and its name. A LoopExp instance can be either an IterateExp instance or an IteratorExp instance. An IterateExp instance represents an *iterate* operator, which permits using an accumulator variable *result* during the iteration process over the *source* collection. An IteratorExp instance permits using the predefined collection iterators: *select*, *reject*, *any*, *sortedBy*, *collect*, *collectNested*, *one*, *forAll*, *exists*, and *isUnique*.
- LiteralExp: A LiteralExp instance is an expression with no arguments producing a value. This includes values like the integer 1 or literal strings like 'this is a LiteralExp.'
- OperationCallExp: An OperationCallExp instance refers to an operation that is defined in an object type. In our approach, we only consider the operators of the object types that have been defined in the metamodel definition $\widetilde{\mathrm{OclStdLib}}$, that is, the predefined operators of the OCL language. An OperationCallExp instance may contain a list of *argument* expressions if the operation that is referred to has parameters. In this case, the number and types of the arguments must match the parameters.
- PropertyCallExp: A PropertyCallExp instance is a reference to a property that is defined in an object type in $\widetilde{\mathcal{M}}$. It evaluates to the value of the attribute.
- VariableExp: A VariableExp instance represents an expression that consists of a reference to a variable. The variables that can be referenced are those that can be defined in the environment of an OCL expression.
- TypeExp: A TypeExp instance permits referring to meta-types, which are defined in a metamodel definition $\widetilde{\mathcal{M}}$, in an OCL expression. In particular, the object type TypeExp permits defining an OCL expression where any of the following operators is used: *allInstances*, *oclIsKindOf*, *oclIsTypeOf*, or *oclAsType*.

7.2 Algebraic Semantics of OCL Predefined Operators

Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the $reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory includes the $reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory, i.e., $reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}) \subseteq reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$, preserving the semantics of the $[\![\mathcal{M}]\!]_{\mathrm{MOF}}$ model type. The

$reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory defines the constrained model type $[\![(\mathcal{M}, \mathcal{C})]\!]_{\text{MOF}}$ by means of a membership axiom of the form

$$\widetilde{M} : \mathcal{M} \;\wedge\; condition_1(\widetilde{M}) = true \;\wedge \cdots \wedge\; condition_n(\widetilde{M}) = true \Longrightarrow \widetilde{M} : (\mathcal{M}, \mathcal{C}),$$

where each constraint definition $\widetilde{c_i}$, in $\widetilde{\mathcal{C}} = \{\widetilde{c_1}, ..., \widetilde{c_n}\}$, corresponds to a function

$$condition_i : [\![\mathcal{M}]\!]_{\text{MOF}} \longrightarrow [\![\text{Boolean}]\!]_{\text{MOF}}.$$

A constraint definition $\widetilde{c}$, such that $\widetilde{c}$: OCL and $root(\widetilde{c}, \widetilde{\text{OCL}})$: Constraint, is a user-defined OCL expression where a property (Property instance) that is defined for an object type in $\widetilde{\mathcal{M}}$ can be referenced by means of a PropertyCallExp instance; and an operation[3] that is predefined in the metamodel definition $\widetilde{\text{OclStdLib}}$ can be referenced by means of an OperationCallExp instance or a LoopExp instance. On the one hand, the OCL predefined operators, which are provided as data in $\widetilde{\text{OclStdLib}}$, are algebraically defined in the parameterized theories `OCL-COLLECTIONS{T :: TRIV}` and `MODEL{OBJ :: TH-OBJECT}`, shown in Fig. 7.6, which are instantiated in the theory $reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory. On the other hand, the *reflect* function maps user-defined OCL expressions to sorts, operators and equations in the $reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory in order to define the corresponding $condition_i$ operator.

In this section, we present the algebraic definition of the OCL predefined operators by extending the structure of parameterized theories that is defined in Section 6. The theories that are modified or added to the existing infrastructure of parameterized theories, which are depicted with a dashed background in Fig. 7.6, are:

- `OCL-BOOL`. The `OCL-BOOL` theory redefines the operators `and`, `or` and `implies` that are defined in the `BOOL` theory, in order to provide the semantics that is defined for these operators in the OCL specification.
- `ENVIRONMENT`. The environment of an OCL expression provides access to model elements or values that are needed for the evaluation of the OCL expression. The `ENVIRONMENT` theory provides the sorts and operators that permit defining the environment of an OCL expression in the $reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory.
- `OCL-COLLECTIONS{T :: TRIV}`. This theory includes the `OCL-COLLECTIONS-TYPES{T :: TRIV}` theory, preserving the semantics of the parameterized OCL types. While the `OCL-COLLECTIONS-TYPES{T :: TRIV}` theory defines the parameterized OCL collection types, the `OCL-COLLECTIONS{T :: TRIV}` provides the predefined operators that are defined, as data, for these types in the metamodel definition $\widetilde{\text{OclStdLib}}$. These operators involve:
  - common operators to all of the types: `=`, `<>`, `oclIsUndefined`;
  - regular operators for collection types;
  - loop operators for collection types;
  - and operators for the special types *OclAny* and *OclType*.
- `MODEL{OBJ :: TH-OBJECT}`. This theory is extended with operators that are defined for object types in the metamodel definition $\widetilde{\text{OclStdLib}}$. More specifically, the operators that permit querying properties in objects are added to this theory.

Considering this extended infrastructure of MEL theories, when the $reflect_{MOF}$ function processes a metamodel definition $\widetilde{\mathcal{M}}$, the resulting $reflect_{MOF}(\widetilde{\mathcal{M}})$ theory does not only provide the semantics of the types that are defined as data in $\widetilde{\mathcal{M}}$ but also the predefined operators that are defined for them in the metamodel definition $\widetilde{\text{OclStdLib}}$. However, we have delayed their presentation until now for the sake of simplicity. In subsequent paragraphs, the sorts and operators that are added to the theories, which are depicted with dashed background in Fig. 7.6, are presented in detail.

7.2.1 Primitive Type Theories

The semantics of the four primitive types, Boolean, String, Integer, and Real, is provided in Section 6.1. The operators that are defined for each one of these types in the OCL specification are already defined in the corresponding MEL theory for each basic type, i.e., `BOOL`, `STRING`, `INT` and `FLOAT`, respectively. In Table 7.1, we show the correspondences between OCL 2.0 and the Maude data-type system and their corresponding operators. In the table, when the operators have different symbols in OCL and Maude, we indicate the

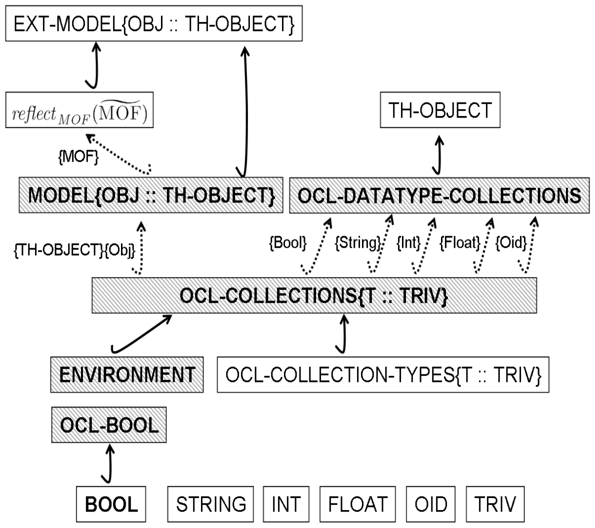


Figure 7.6: Parameterized theories that provide the predefined OCL operators.

OCL 2.0	Maude	Operators
Boolean	Bool	$= (\_{=}\_)$, $<> (\_{<>}\_)$, $or(\_or\_)$, $and(\_and\_)$, $xor(\_xor\_)$, $not(not\_)$, $implies(\_implies\_)$,
String	String	$= (\_{=}\_)$, $<> (\_{<>}\_)$, $size(length)$, $concat(\_{+}\_)$, $substring(substr)$, $(find)$, $(rfind)$, $(\_{<}\_)$, $(\_{<=}\_)$, $(\_{>}\_)$, $(\_{>=}\_)$
Integer	Int	$= (\_ = \_)$, $<> (\_{<>}\_)$, $+(\_{+}\_)$, $-(\_{-}\_)$, $*(\_{*}\_)$, $unary-(\_{-})$, $/(\_quo\_)$, $abs(abs)$, $div(\_div\_)$, $mod(\_mod\_)$, $floor(floor)$, $round(round)$, $max(max)$, $min(min)$, $< (\_ < \_)$, $<= (\_{<=}\_)$, $> (\_{>}\_)$, $>= (\_{>=}\_)$
Real	Float	$= (\_{=}\_)$, $<> (\_{<>}\_)$, $+(\_{+}\_)$, $-(\_{-}\_)$, $*(\_{*}\_)$, $unary-(\_{-})$, $/(\_quo\_)$, $abs(abs)$, $floor(floor)$, $round(ceiling)$, $max(max)$, $min(min)$, $< (\_ < \_)$, $<= (\_{<=}\_)$, $> (\_{>}\_)$, $>= (\_{>=}\_)$

Table 7.1: Correspondence between MOF and Maude basic data type operators.

b_1	b_2	b_1 *and* b_2	b_1 *or* b_2	b_1 *implies* b_2
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	true
false	$\perp$	false	$\perp$	true
true	$\perp$	$\perp$	true	$\perp$
$\perp$	false	false	$\perp$	$\perp$
$\perp$	true	$\perp$	true	true
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Table 7.2: Semantics of boolean operators.

Maude symbol in parentheses. In addition, we reuse some Maude basic data type operators that are not included in the standard, which are shown in parentheses.

While in the OCL specification, Integer is a subtype of Real, in the `INT` and `FLOAT` theories, these two types are represented by *Int* and *Float* sorts, respectively. These sorts belong to different kinds and, therefore, they are not related by means of a subsort relationship. Defining a supersort of both, like the *OclAny* type in OCL, would produce name collisions for the operator symbols. For example, $\_+\_ : Int\ Int \rightarrow Int$ and $\_+\_ : Float\ Float \rightarrow Float$ are defined and both of them cannot coexist as they are, unless *Int* and *Float* sorts remain in different kinds. An operator renaming would make this coexistence feasible but we have chosen the first approach: to keep the original symbols.

Some boolean operators of the `BOOL` theory, namely *and*, *or* and *implies*, constitute an exception in order to manage with undefined values. We follow the semantics that is provided in [32] for these operators, as shown in Table 7.2. These boolean operators are redefined in the `OCL-BOOL` theory, which is presented, in Maude notation, as

```
fmod OCL-BOOL is
    protecting BOOL * (
        op and to maudeAnd,
        op or to maudeOr,
        op implies to maudeImplies
    ) .

    vars A B : Bool .
    vars undefA undefB : [Bool] .

    op _and_ : Bool Bool ~> Bool [assoc comm] .
    eq false and undefB = false .
    eq A and B = A maudeAnd B .

    op _or_ : Bool Bool ~> Bool [assoc comm] .
    eq true or undefB = true .
    eq A or B = A maudeOr B .

    op _implies_ : Bool Bool ~> Bool .
    ceq false implies undefB = true
    if not(undefB :: Bool) .
    ceq undefA implies true = true
    if not(undefA :: Bool) .
    eq A implies B = A maudeImplies B .
endfm
```

where `maudeAnd`, `maudeOr` and `maudeImplies` represent the original boolean maude operators *and*, *or* and *implies*, which have been renamed. The `Bool` view is, then, redefined as follows:

[3]In our approach, we do not consider operations that are defined for object types in $\widetilde{\mathcal{M}}$.

```
view Bool from TRIV to OCL-BOOL is
    sort Elt to Bool .
endv
```

7.2.2 The ENVIRONMENT Theory

The environment of an OCL expression defines what model elements are visible and can be referred to in the expression. As shown in Section 7.1, OCL constraints are defined as collections of nested OclExpression instances in a model definition $\widetilde{c}$, such that $\widetilde{c}$: OCL and $root(\widetilde{c}, \widetilde{\mathrm{OCL}})$: Constraint, where the nesting relation corresponds to the containment relation $<_c$ that is defined as the set $<_c (\widetilde{c}, \widetilde{\mathrm{OCL}})$. We define the set of types D' as

$$\mathrm{D}' = \{ \texttt{Bool, String, Int, Float, Oid, Enum, Object\#}\mathcal{M}\}$$

, and the set D as

$$\begin{aligned}\mathrm{D} = \bigcup_{\mathrm{T} \in \mathrm{D}'} & \{ \texttt{NeSet\{T\}, Set\{T\}, NeOrderedSet\{T\}, OrderedSet\{T\},} \\ & \texttt{NeBag\{T\}, Bag\{T\}, NeSequence\{T\}, Sequence\{T\},} \\ & \texttt{Collection\{T\}} \} \\ \cup\ \mathrm{D}' &\end{aligned}$$

where `T` is a view name that corresponds to the sorts in D'. The set of values that can be used to define objects in a specific model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, is defined as

$$\mathcal{D} = \bigcup_{t \in D} (T_{reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}), t}).$$

An OCL variable is a pair of the form *(name = value)*, where $name \in OpNames$ and $value \in \mathcal{D}$. The sorts and constructors that permit defining variables in the environment of an OCL expression are defined in the `ENVIRONMENT` theory, in Maude notation, as follows:

```
fmod ENVIRONMENT is
    sorts VariableName Variable Environment .
    subsort Variable < Environment .

    op empty-env : -> Environment .
    op _,_ : Environment Environment -> Environment
        [comm assoc id: empty-env] .
endfm
```

This theory is sufficient to define the `OCL-COLLECTIONS{T :: TRIV}` theory. However, we cannot define variables yet. The operators that permit both defining variables in a term of sort `Environment` and querying variable values are defined in the `OCL-DATATYPE-COLLECTIONS` theory as follows:

```
    op _=_ : VariableName Collection+{Bool} -> Variable .
    op _=_ : VariableName Collection+{String} -> Variable .
    op _=_ : VariableName Collection+{Int} -> Variable .
    op _=_ : VariableName Collection+{Float} -> Variable .
    op _=_ : VariableName Collection+{Enum} -> Variable .
    op _=_ : VariableName Collection+{Oid} -> Variable .

    var VN : VariableName .

    op GetBoolVarValue : Variable ~> Collection+{Bool} .
    eq GetBoolVarValue(VN = V:Collection+{Bool}) =
        V:Collection+{Bool} .

    op GetStringVarValue : Variable ~> Collection+{String} .
    eq GetStringVarValue(VN = V:Collection+{String}) =
        V:Collection+{String} .
```

Return Type	Collection Operator Symbols					Iterator Symbols
	Collection	Set	OrderedSet	Bag	Sequence	Collection
Collection{T}	union, flatten, including, excluding	-, intersection	-, insertAt, append, prepend	intesection	insertAt, append, prepend	select, reject, any, sortedBy, collect, collectNested, iterate
Collection+{T}			first, last, at		first, last, at	
Boolean	includes, includesAll, excludes, excludesAll, isEmpty, notEmpty					one, forAll, forAll2, exists, isUnique
Int	count, size, sum, product		indexOf		indexOf	

Table 7.3: OCL Collection Operators Classified by their Returning Types

```
op GetIntVarValue : Variable ~> Collection+{Int} .
eq GetIntVarValue(VN = V:Collection+{Int}) =
    V:Collection+{Int} .

op GetFloatVarValue : Variable ~> Collection+{Float} .
eq GetFloatVarValue(VN = V:Collection+{Float}) =
    V:Collection+{Float} .

op GetEnumVarValue : Variable ~> Collection+{Enum} .
eq GetEnumVarValue(VN = V:Collection+{Enum}) =
    V:Collection+{Enum} .

op GetOidVarValue : Variable ~> Collection+{Oid} .
eq GetOidVarValue(VN = V:Collection+{Oid}) =
    V:Collection+{Oid} .
```

In the *reflect*($\widetilde{\mathcal{M}}$, $\widetilde{\mathcal{C}}$) theory, an OCL variable can be defined by means of the expression `(nameValue : "Table")`, where `nameValue : VariableName`.

7.2.3 The `OCL-COLLECTIONS{T :: TRIV}` Theory

This theory provides the equational definition of the OCL operators that are defined as data in the metamodel definition $\widetilde{\mathrm{OclStdLib}}$: common operators to all OCL types (=, <> and *oclIsUndefined*), and predefined operators for OCL collection types. Two kinds of collection operators can be distinguished in OCL 2.0: *regular operators*, which provide common functionality over collections, such as the `size` operator that computes the cardinality of a given collection; and *loop operators*, which permit iterating over the elements in a source collection performing a specific action, such as `forAll`, `sortedBy` and `iterate`. Loop operators can be classified in *iterator operators*, which do not permit accumulating a value while the source collection is traversed, and the *iterate operator*, which is the most general loop operator and does provide this functionality. The collection operators that are supported in our specification are shown in Table 7.3, classified by the type of the source collection, to which they can be applied (columns), and by their returning types (rows).

Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the `OCL-COLLECTIONS{T :: TRIV}` theory is instantiated for the primitive types, enumeration types and object types that are defined in $\widetilde{\mathcal{M}}$ in the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory. To define collections of objects $\widetilde{o}$, such that $\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),Object\#\mathcal{M}}$, the `OCL-COLLECTIONS{T :: TRIV}` theory is instantiated with the `TH-OBJECT`$\{\mathcal{M}\}$ view, which is defined, in Maude notation, as follows:

```
view TH-OBJECT from TRIV to TH-OBJECT is
    sort Elt to Object .
endv
```

Therefore, the `OCL-COLLECTIONS{T :: TRIV}` theory can be instantiated for a specific metamodel $\mathcal{M}$ by means of the expression `OCL-COLLECTIONS{TH-OBJECT}`$\{\mathcal{M}\}$, as shown in Fig. 7.6.

To illustrate the `OCL-COLLECTIONS{T :: TRIV}` theory, we present the algebraic definition of some operators: a regular collection operator, an iterator operator and the *iterate* operator. The complete specification of the OCL collection operators is provided in Appendix F.

Common Operators

The `OCL-COLLECTIONS{T :: TRIV}` theory provides the so-called *common operators* (=, <>, and *oclIsUndefined*) for OCL types. These operators are defined as follows:

- (_ = _) : This operator checks if two values are the same. The operator can also be applied to collections of values. The operator (_ = _) is defined, in Maude notation, as follows:

  ```
  vars E1 E2 : Collection+{T} .
  vars undef1 undef2 : [Collection+{T}] .

  op _=_ : [Collection+{T}] [Collection+{T}] -> Bool .
  eq (undef1 = undef2) = not(undef1 :: Collection+{T})
      and not(undef2 :: Collection+{T}) [owise] .
  ```

  This operator is not completely defined in this theory because to define the equality between objects we need information that we do not have yet in this theory, like object identifiers. The complete definition of this operator is achieved in the `MODEL{OBJ :: TH-OBJECT}` theory.

- (_ <> _) : This operator is defined as the negation of the previous one:

  ```
  op _<>_ : [Collection+{T}] [Collection+{T}] -> Bool .
  eq (undef1 <> undef2) = not(undef1 = undef2) .
  ```

- (_ .*oclIsUndefined*) : This operator checks wether a value is undefined or not. The operator is defined, in Maude notation, as follows:

  ```
  op _.oclIsUndefined : [Collection+{T}] -> Bool .
  eq undef1 . oclIsUndefined = not(undef1 :: Collection+{T}) .
  ```

  Recall that the generation of an undefined value for each one of the primitive types is achieved by means of the corresponding constructor `op nullBool : -> [Bool]`, `op nullString : -> [String]`, `op nullInt : -> [Int]`, and `op nullFloat : -> [Float]`.

Regular Collection Operators

In the `OCL-COLLECTIONS{T :: TRIV}` theory, the EBNF rules of the OCL grammar have been specified as operators in mixfix notation. Each OCL expression, in which a collection operator is invoked, is represented as a term by means of operators of this kind. These terms are written in a format close to the concrete syntax of OCL due to the mixfix notation of the operators. For example, to invoke the operator `size` we provide the following operator:

```
op _->`size : Collection{T} -> Int .
```

where the argument of the operator is the source collection over which the operator is evaluated. For example, the term `Set{1,2} -> size` represents an OCL expression that computes the size of a source set of integers.

The semantics of the predefined OCL operators is provided in the form of equations. These equations can be interpreted from two points of view[4]: from a programming point of view, the equations provide the operational semantics of the operators in a functional program; and from a specification point of view, the equations, together with the sorts and operators, provide a MEL theory with initial algebra semantics. Taking into account the semantics that are defined in [93], the following set of equations defines how to evaluate an invocation of the `size` operator over a set of elements:

```
var N : Collection+{T} .
var M : Magma{T} .

eq Set{ N, M } -> size = Set{ M } + 1 .
eq Set{ N } -> size = 1 .
eq empty-set -> size = 0 .
```

Note that the function `size` is only defined when the argument collection is not an undefined value.

Iterator Operators

Operators of this kind receive an OCL expression as argument and execute it over each element of a source collection. The definition of an iterator operator can be split in several parts:

1. Operator name: represents the name of the specific operator. We consider the operators *select*, *reject*, *any*, *sortedBy*, *collect*, *collectNested*, *one*, *forAll*, *exists*, and *isUnique*.
2. Source collection: collection of elements that is traversed.
3. Body expression: represents the expression that is evaluated for each of the elements of the source collection by the iterator operator. Body expressions are defined as functions whose evaluation results in a valid OCL type value. A body expression is identified by a symbol, which is a term of one of the following sorts:

   - `Body{T}` : represents the name of a body expression whose evaluation returns a term of sort `Collection+{T}`;
   - `BoolBody{T}` : represents the name of a body expression whose evaluation returns a term of sort `Collection+{Bool}`;
   - `StringBody{T}` : represents the name of a body expression whose evaluation returns a term of sort `Collection+{String}`;
   - `IntBody{T}` : represents the name of a body expression whose evaluation returns a term of sort `Collection+{Int}`;
   - `FloatBody{T}` : represents the name of a body expression whose evaluation returns a term of sort `Collection+{Float}`; and
   - `EnumBody{T}` : represents the name of a body expression whose evaluation returns a term of sort `Collection+{Enum}`.

   These sorts are used in constructors that permit defining a body expression as follows:

   ```
   op _._(_;_) : Magma{T} Body{T} Environment
       PreConfiguration{T} -> Collection+{Oid} .
   op _._(_;_) : Magma{T} BoolBody{T} Environment
       PreConfiguration{T} -> Collection+{Bool} .
   op _._(_;_) : Magma{T} StringBody{T} Environment
       PreConfiguration{T} -> Collection+{String} .
   op _._(_;_) : Magma{T} IntBody{T} Environment
       PreConfiguration{T} -> Collection+{Int} .
   op _._(_;_) : Magma{T} FloatBody{T} Environment
   ```

[4]We assume that the theories that we provide satisfy the executability requirements for a MEL theory. However, we do not provide a formal proof of the satisfaction of these requirements.

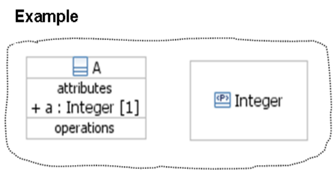


Figure 7.7: Metamodel definition $\widetilde{\text{EXAMPLE}}$ as a class diagram.

```
        Configuration{T} -> Collection+{Float} .
    op _._(_;_) : Magma{T} EnumBody{T} Environment
        PreConfiguration{T} -> Collection+{Enum} .
```

where: (i) the first argument represents the element of the source collection, to which the body expression is applied in a step of the iteration process; (ii) the second argument represents the symbol of the body expression; (iii) the third argument is the environment for the body expression, which provides a set of variables that can be used in the evaluation of the body expression; and (iv) the fourth argument is a term of sort `PreConfiguration{T}`, whose constructor is defined as

```
op nonePreConf#T : -> PreConfiguration{T} .
```

An OCL expression that is applied to the objects of a source collection may involve a navigation through object-typed properties in a model definition $\widetilde{M}$. Therefore, the entire model definition $\widetilde{M}$ is needed to evaluate the OCL expression. In the `MODEL{OBJ :: TH-OBJECT}` theory, the sort `PreConfiguration{TH-OBJECT}{OBJ}` is defined as supersort of `Configuration{OBJ}`, so that a model definition $\widetilde{M}$ is also a term of sort `PreConfiguration{TH-OBJECT}{OBJ}`. When the source collection that is traversed is a collection of DATATYPE values, i.e., primitive type values or enumeration literals, the fourth argument is not needed. In these cases, the constant `nonePreConf#T` is used.

4. Environment: the environment for an iterator operator is a set of variables that may have been defined in the context of the OCL expression, in a *let* statement or in other *loop* operators. In a specific *loop* operator, the *iterator variable* is used to refer to the element of the source collection that is being traversed in each step of the iteration process. The iterator variable value corresponds to the first argument of the operator `_._(_;_)` for a body expression.

5. Model definition: the model definition $\widetilde{M}$ that contains the source collection of elements, when it is a collection of objects. A term of sort `PreConfiguration{T}` is used in these cases. When the source collection that is traversed is a collection of DATATYPE values, i.e., primitive type values or enumeration literals, this argument is not needed.

To illustrate how the semantics of the OCL iterator operators has been equationally-defined in the `OCL-COLLECTIONS{T :: TRIV}` theory, we provide the algebraic specification of the `forAll` and `sortedBy` operators. The algebraic specification for the rest of collection operators is given in Appendix F. Consider the metamodel definition $\widetilde{\text{EXAMPLE}}$, shown in Fig. 7.7 as a class diagram that contains the definition of the A object type. This object type is defined with a single property, named a, of type INTEGER. We define a model definition $\widetilde{M}$, such that $\widetilde{M}$: EXAMPLE as the collection of objects

```
<<
    < oid#A('Foo1) : A | a : 3 >
    < oid#A('Foo2) : A | a : 2 >
    < oid#A('Foo3) : A | a : 5 >
    < oid#A('Foo4) : A | a : 1 >
>>.
```

The `forAll` operator represents a universal quantifier that checks whether each element in the source collection satisfies a given condition or not. As a guiding example, we use an expression, using the concrete syntax of the OCL language, that indicates whether all the numbers in a set of integers are odd or not:

```
A.allInstances() -> forAll(objA : A | objA.a.mod(2) <> 0)
```

where `A.allInstances()` obtains all instances of the class A in the model definition $\widetilde{M}$ as a set of objects [5]. In this expression, `forAll` is the iterator operator, `A.allInstances()` provides the source collection, `objA : A` is the iterator variable, and `(objA.a.mod(2) <> 0)` is the body expression. Using the example of the condition of odd numbers, we study first how to specify the forAll body expression `objA : A | objA.a.mod(2) <> 0`.

Body expressions are defined as functions whose evaluation results in a valid OCL type value. The body expression of the `forAll` operator evaluates to a boolean value. It is defined by a constant of the sort `BoolBody{Example}` as follows:

```
op isOdd : -> BoolBody{Example} .
```

For the example, the body expression of the `forAll` operator is provided, in Maude notation, as

```
eq objA:Object#Example . isOdd (
    empty-env ;
    model:ModelType{Example}
) =
    (((objA:Object#Example . a) rem 2) =/= 0) .
```

where `E` represents the environment of the body expression, and `a` is an operator defined as `op a : -> IntFun{Example}`, which permits obtaining the value of the property `a` of an instance of the object type A, as explained below. The syntax of the `forAll` expression is defined by the operator

```
op _->‘forAll‘(_;_;_‘) : Collection{T} BoolBody{T} Environment
    PreConfiguration{T} -> Bool .
```

where the first argument is the source collection to be traversed, the second argument is the symbol that identifies the body expression (`isOdd` for the example), the third argument is the environment of the body expression, and the fourth argument is a term of sort `PreConfiguration{T}`, which can be a model definition $\widetilde{M}$, such that $\widetilde{M}$: EXAMPLE in the example, when the source collection is a collection of objects, as in the example.

The semantics of iterator operators is defined generically simulating higher-order functions. Three equations constitute the algebraic specification of the *forAll* operator for sets in the `OCL-COLLECTIONS{T :: TRIV}` theory:

```
var N : Collection+{T} .
var M : Magma{T} .
var BB : BoolBody{T} .
var E : Environment .
var PR : PreCofiguration{T} .

eq Set{ N , M } -> forAll (  BB ; E ; PR ) =
    (N . BB ( E ; PR )) and (Set{ M } -> forAll ( BB ; E ; PR )) .
eq Set{ N } -> forAll ( BB ; E ; PR ) = N . BB ( E ; PR ) .
eq empty-set -> forAll ( BB ; E ; PR ) = true .
```

The first equation considers the recursion case where there is more than one element in the set. The second equation considers the recursion case when only one element remains in the set so that the recursive trail ends. The third equation considers the case where the set is empty. To invoke the `forAll` iterator over a set of objects with the body `isOdd` we use the term:

$$\texttt{A.allInstances(}\widetilde{M}\texttt{)-> forAll(isOdd ; empty-env ; }\widetilde{M}\texttt{).}$$

Among the OCL iterator operators, we find the *sortedBy* operator, which permits ordering the elements of a given collection. For example, given the model definition $\widetilde{M}$, such that $\widetilde{M}$: RDBMS, that is represented in Fig. 7.8, the expression

```
Schema.allInstances()
    -> sortedBy( t : Table | t.name )
    -> collect( t : Table | t.name),
```

[5]An equational definition of the `allInstances` operator is given below.

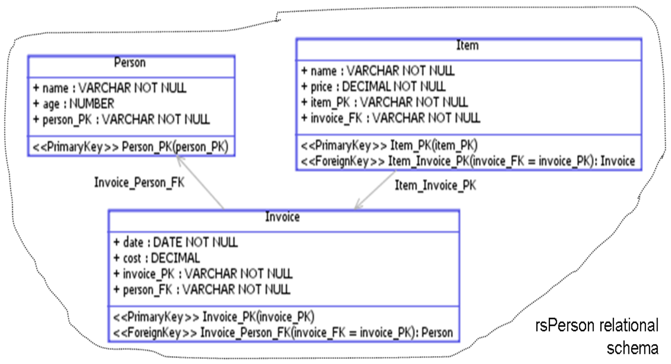

Person
+ name : VARCHAR NOT NULL
+ age : NUMBER
+ person_PK : VARCHAR NOT NULL
<<PrimaryKey>> Person_PK(person_PK)
Item
+ name : VARCHAR NOT NULL
+ price : DECIMAL NOT NULL
+ item_PK : VARCHAR NOT NULL
+ invoice_FK : VARCHAR NOT NULL
<<PrimaryKey>> Item_PK(item_PK)
<<ForeignKey>> Item_Invoice_PK(invoice_FK = invoice_PK): Invoice
Invoice_Person_FK
Item_Invoice_PK
Invoice
+ date : DATE NOT NULL
+ cost : DECIMAL
+ invoice_PK : VARCHAR NOT NULL
+ person_FK : VARCHAR NOT NULL
<<PrimaryKey>> Invoice_PK(invoice_PK)
<<ForeignKey>> Invoice_Person_FK(invoice_FK = invoice_PK): Person
rsPerson relational schema

Figure 7.8: A relational schema.

which is presented using the OCL textual formal, results in an ordered set `OrderedSet{'Invoice', 'Item', 'Person'}` (also represented in OCL textual format). The parameter of this operation is a property of the object type of the elements in the collection. Also the elements themselves can be used as parameter. For example, the expression `Set{2,1} -> sortedBy( i : Integer | i )` results in the value `OrderedSet{1,2}`. For the type of the parameter, the *lesserThan* operation (also denoted by <) must be defined. The operator *sortedBy* loops over all elements in the *source* collection and orders all elements according to the parameter value. The first element in the result is the element for which the parameter value is the lowest.

The `sortedBy` operator is important in our specification because it permits ordering an unsorted collection in a deterministic way. We define the *lesserThan* operation as a boolean body expression by means of equations of the form

```
eq N1:T . lesserThanString (
    VN:VariableName = N2:T ; pr:PreConfiguration{T} ) = ...
```

where `N1:T` and `N2:T` are the elements to be compared. The *lesserThan* operator is defined for the sorts that represent primitive types, (`String`, `Int` and `Float`), for object types (`Object`), and for object identifier types (`Oid`), in the `MODEL{OBJ :: TH-OBJECT}` theory as follows:

```
op lesserThanString : -> BoolBody{String} .
eq N1:String . lesserThanString
    ( VN:VariableName = N2:String ; pr:PreConfiguration{String} ) =
    (N1:String < N2:String) .

op lesserThanInt : -> BoolBody{Int} .
eq N1:Int . lesserThanInt
    ( VN:VariableName = N2:Int ; pr:PreConfiguration{Int} ) =
    (N1:Int < N2:Int) .

op lesserThanFloat : -> BoolBody{Float} .
eq N1:Float . lesserThanFloat
    ( VN:VariableName = N2:Float ; pr:PreConfiguration{Float} ) =
    (N1:Float < N2:Float) .

op lesserThanOid : -> BoolBody{Oid} .
eq N1:Oid . lesserThanOid
    ( VN:VariableName = N2:Oid ; pr:PreConfiguration{Oid} ) =
    (string(N1:Oid) < string(N2:Oid)) .

op lesserThanObject : -> BoolBody{TH-OBJECT}{OBJ} .
eq N1:Object#OBJ . lesserThanObject
    ( VN:VariableName = N2:Object#OBJ ;
    pr:PreConfiguration{TH-OBJECT}{OBJ} ) =
    oid(N1:Object#OBJ) . lesserThanOid(
        VN:VariableName = oid(N2:Object#OBJ) ; nonePrConf#Oid
    ) .
```

where *string* : *Oid* $\longrightarrow$ *String* is a function that obtains a string from an object identifier.

The `sortedBy` operator is defined in the `OCL-COLLECTIONS{T :: TRIV}` theory for the case where the parameter of the operator is the iterator variable, and in the `MODEL{OBJ :: TH-OBJECT}` theory for the case where the source collection is a collection of objects and the parameter of the `sortedBy` operator is a property of an object.

In the first case, this operator is defined in the `OCL-COLLECTIONS{T :: TRIV}` theory, in Maude notation, as

```
op _->`sortedBy`(_;_`) :
    Collection{T} BoolBody{T} Environment -> Collection{T} .
```

where: the first argument is the source collection to be ordered; the second argument is an operator name that refers to a *lesserThan* operator; and the third argument is the environment of variables that can be used in the *lesserThan* operator. The OCL expression, in OCL textual concrete syntax, `Set{3,2} -> sortedBy( i : Integer | i )`, is represented by the term `Set{3,2} -> sortedBy( lesserThanInt ; empty-env )`, which is reduced to the canonical form `OrderedSet{2 :: 3}`.

In the second case, the `sortedBy` operator is defined in the `MODEL{OBJ :: TH-OBJECT}` theory for each type of object property that can be used as parameter. We consider the types `String`, `Int`, `Float`, and `Oid` by means of the operators:

```
op _->`sortedBy`(_;_;_`) :
    Collection{TH-OBJECT}{OBJ} StringBody{TH-OBJECT}{OBJ}
    Environment PreConfiguration{TH-OBJECT}{OBJ}
    -> Collection{TH-OBJECT}{OBJ} .

op _->`sortedBy`(_;_;_`) :
    Collection{TH-OBJECT}{OBJ} IntBody{TH-OBJECT}{OBJ}
    Environment PreConfiguration{TH-OBJECT}{OBJ}
    -> Collection{TH-OBJECT}{OBJ} .

op _->`sortedBy`(_;_;_`) :
    Collection{TH-OBJECT}{OBJ} FloatBody{TH-OBJECT}{OBJ}
    Environment PreConfiguration{TH-OBJECT}{OBJ}
    -> Collection{TH-OBJECT}{OBJ} .

op _->`sortedBy`(_;_;_`) :
    Collection{TH-OBJECT}{OBJ} Body{TH-OBJECT}{OBJ}
    Environment PreConfiguration{TH-OBJECT}{OBJ}
    -> Collection{TH-OBJECT}{OBJ} .
```

where: the first argument is the source collection to be ordered; the second argument is an operator name that identifies the body expression that is used to obtain the ordering value; the third argument is the environment of variables that can be used in the body expression, and the fourth argument is the model definition so that navigations through object-typed properties are also allowed in the body expression.

The OCL expression, in OCL textual concrete syntax, `Schema.allInstances() -> sortedBy( t : Table | t.name )`, is represented by the term `Schema.allInstances(` $\widetilde{M}$ `) -> sortedBy( getName ; empty-env ;` $\widetilde{M}$ `)`, where the `getName` body expression is defined as follows:

```
op getName : -> StringBody{rdbms} .
eq tValue:Collection+{rdbms} . getName(
    empty-env ; model:ModelType{rdbms}
) = tValue:Collection+{rdbms} . name .
```

Iterate Operator

The *iterate* operator is the most generic loop operation. All other loop operations can be described as special cases of *iterate*, as shown in [24]. The concrete syntax of the iterate operation is as follows:

```
    collection -> iterate( element : Type1;
        result  : Type2 = <expression>
        | <expression-with-element-and-result>)
```

The variable `element` is the iterator variable. The resulting value is accumulated in the variable `result`, which is also called the *accumulator* variable. The accumulator variable gets an initial value, given by the `<expression>` after the equal sign. None of the parameters is optional.

The result of the iterate operation is a value obtained by iterating over all elements in a collection. For each successive element in the source collection, the body expression `<expression-with-element-and-result>` is calculated using the previous value of the result variable. A simple example of the iterate operation is given by the following expression, which results in the sum of the elements of a set of integers:

```
Set{1, 2, 3} -> iterate( i:  Integer, sum:  Integer = 0 | sum + i ).
```

The iterate operator is declared in the `OCL-COLLECTIONS{T :: TRIV}` theory as:

```
op _->`iterate`(_|_;_;_`) :
    Collection{T} Variable IterateBody{T} Environment PreConfiguration{T}
    ~> Variable .
```

where: the first argument is the source collection to be traversed; the second argument is the symbol that is associated to the iterate body expression; the third argument is the *result* variable; the fourth argument is the environment that may be used to evaluate the iterate body expression; the fifth argument is the model definition $\widetilde{M}$, which contains the objects that are included in the source collection, when the source collection is a collection of objects (in any other case, this argument is not needed). The iterate operator returns a term of sort `Variable`, which represents the *accumulator* variable.

When the source collection is a set of elements, the iterate operator is defined, in Maude notation, by means of the following equations:

```
var N : Collection+{T} .
var M : Magma{T} .
var result : Variable .
var IF : IterateBody{T} .
var E : Environment .
var Pr : PreConfiguration{T} .

eq Set{ N , M } -> iterate ( result | IF ; E ; Pr ) =
    N . IF ( (Set{ M } -> iterate ( result | IF ; E ; Pr )) ; PL ; Pr ) .
eq Set{ N } -> iterate ( result| IF ; E ; Pr ) =
    N . IF ( result ; E ; Pr ) .
eq empty-set -> iterate ( result | IF ; E ; Pr ) = result.
```

where the variable `IF` represents the symbol that identifies an iterate body expression. An iterate body expression is defined by means of the operator:

```
op _._`(_;_;_`) :
    Collection+{T} IterateBody{T} Variable Environment PreConfiguration{T}
    ~> Variable .
```

where: the first argument is the value of the *iterator* variable in a specific iteration step of the iterate operator; the second argument is the symbol that is associated to the iterate body expression; the third argument is the *accumulator* variable; the fourth argument is the environment that may be used to evaluate the iterate body expression; the fifth argument is the model definition $\widetilde{M}$, which contains the objects that are included in the source collection, when the source collection is a collection of objects (in any other case, this argument is not needed). An iterate body expression returns a term of sort `Variable`, which represents the *accumulator* variable. The iterate body expression that provides the algebraic semantics for the body expression `(i: Integer, sum: Integer = 0 | sum + i )` is algebraically defined by means of the following equation:

```
op sum : -> VariableName .
op integerSum : -> iterateBody{Int} .

eq iValue:Collection+{Int} . integerSum(
    sum = sumValue:Collection+{Int} ; empty-env ; nonePrConf#Int) =
        (sum = (sumValue:Collection+{Int} + iValue:Collection+{Int})) .
```

Therefore, given a set `Set{1,2,3}` of integers, the OCL expression, in OCL concrete syntax,

```
Set{1,2,3} ->iterate ( i: Integer, sum: Integer = 0 | sum + i )
```

is represented by the term

```
Set{1,2,3} -> iterate ( result = 0 | integerSum ; empty-env ; nonePrConf#Int )
```

where `Set{1,2,3}` is a term of sort `Set{Int}`, `empty-env` represents an empty environment, and the constant `nonePrConf#Int` indicates that the last argument is not given since it is not needed. The previous term can be simplified by applying the equations that have been previously defined modulo associativity and commutativity, resulting in the term `result = 6` of sort `Variable`, which represents the sum of all of the integer values of the source set.

7.2.4 The `MODEL{OBJ :: TH-OBJECT}` Theory

Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the `MODEL{OBJ :: TH-OBJECT}` theory provides the operators that are predefined for the object types that are defined in $\widetilde{\mathcal{M}}$. These predefined operators can be classified as: *common* operators, and *user-defined* operators. Among the *user-defined* ones, we can make another classification:

- *Value-typed property operators:* An operator of this type projects the value of a value-typed property.
- *Object-typed property operators:* An operator of this type projects the value of an object-typed property. These operators permit the navigation through the edges of the graph *graph*$(\widetilde{M}, \widetilde{\mathcal{M}})$, where $\widetilde{M} : \mathcal{M}$.
- *Query operators:* These operators are defined as OCL expressions for a specific object type and do not have side effects. The semantics of an object operation is therefore given by the semantics of the associated OCL expression. We do not consider operators of this kind in our approach at the moment.

In addition, the special types *OclAny* and *OclType* of the OCL language provide some operators that are also specified in the `MODEL{OBJ :: TH-OBJECT}` theory.

Common Operations

These operators are defined in the `OCL-COLLECTIONS{T :: TRIV}` theory. However, the operator `=` is redefined to take into account the object identifiers that are intrinsically related to objects. The operator `=` checks if two objects are the same by means of their identifiers. The operator can also be applied to collections of objects. The operator $(\_ = \_)$ is defined, in Maude notation, as follows:

```
vars E1 E2 : Collection+{TH-OBJECT}{OBJ} .
eq (E1 = E2) = (oid(E1) == oid(E2)) .
```

where the operator $oid : Collection\{OBJ\} \rightarrow Collection\{Oid\}$ is an overloaded version of the *oid* operator that obtains a collection of object identifiers from a collection of objects, keeping the order and uniqueness features of each specific domain collection. For example,

```
oid(
    OrderedSet{
        < oid#Class("Class0") : Class | PS1 > ::
        < oid#Class("Class1") : Class | PS2 >
    }
) = OrderedSet{ oid#Class("Class0") :: oid#Class("Class1") }.
```

The `=` operator is also defined for data type values as follows:

```
eq V1:Collection+{Bool} = V2:Collection+{Bool} =
    (V1:Collection+{Bool} == V2:Collection+{Bool}) .
eq V1:Collection+{String} = V2:Collection+{String} =
    (V1:Collection+{String} == V2:Collection+{String}) .
eq V1:Collection+{Int} = V2:Collection+{Int} =
    (V1:Collection+{Int} == V2:Collection+{Int}) .
eq V1:Collection+{Float} = V2:Collection+{Float} =
    (V1:Collection+{Float} == V2:Collection+{Float}) .
eq V1:Collection+{Enum} = V2:Collection+{Enum} =
    (V1:Collection+{Enum} == V2:Collection+{Enum}) .
```

User-defined Operations

To project property values from a specific object term, we defined the following sorts

- `Fun{TH-OBJECT}{OBJ}` : represents the name of property projectors that return a term of sort `Collection+{Oid}`;
- `BoolFun{TH-OBJECT}{OBJ}` : represents the name of property projectors that return a term of sort `Collection+{Bool}`;
- `StringFun{TH-OBJECT}{OBJ}` : represents the name of property projectors that return a term of sort `Collection+{String}`;
- `IntFun{TH-OBJECT}{OBJ}` : represents the name of property projectors that return a term of sort `Collection+{Int}`;
- `FloatFun{TH-OBJECT}{OBJ}` : represents the name of property projectors that return a term of sort `Collection+{Float}`; and
- `EnumFun{TH-OBJECT}{OBJ}` : represents the name of property projectors that return a term of sort `Collection+{Enum}`.

These sorts are used to define the operators that permit querying the value of value-typed properties as follows:

```
op _._ : Collection+{TH-OBJECT}{OBJ} Fun{TH-OBJECT}{OBJ}
    ~> Collection+{Oid} .
op _._ : Collection+{TH-OBJECT}{OBJ} BoolFun{TH-OBJECT}{OBJ}
    ~> Collection+{Bool} .
op _._ : Collection+{TH-OBJECT}{OBJ} StringFun{TH-OBJECT}{OBJ}
    ~> Collection+{String} .
op _._ : Collection+{TH-OBJECT}{OBJ} IntFun{TH-OBJECT}{OBJ}
    ~> Collection+{Int} .
op _._ : Collection+{TH-OBJECT}{OBJ} FloatFun{TH-OBJECT}{OBJ}
    ~> Collection+{Float} .
op _._ : Collection+{TH-OBJECT}{OBJ} EnumFun{TH-OBJECT}{OBJ}
    ~> Collection+{Enum} .
```

For example, in the RDBMS metamodel, to obtain the value of the property name in a table we define the operator `op name : -> StringFun{rdbms}`. The name value of a specific TABLE instance `t` can be then obtained by reducing the term `t.name`. Note that the equation that defines this operator has not been defined yet. The equations that define operators of this kind are generated by means of the *reflect* function, as shown in Section 7.3.

Object-typed properties are those that are defined with an object identifier type. Object-typed properties permit traversing the collection of objects that constitutes a specific model definition $\widetilde{M}$, taking the graph structure $graph(\widetilde{M}, \widetilde{\mathcal{M}})$ into account, by means of the operator

```
op _._(_) : Collection+{TH-OBJECT}{OBJ} Fun{TH-OBJECT}{OBJ}
    Configuration{TH-OBJECT}{OBJ} ~> Collection+{TH-OBJECT}{OBJ} .
```

For example, in the RDBMS metamodel, to obtain the SCHEMA instance that contains a specific TABLE instance `t` in a relational schema $\widetilde{M}$, such that $\widetilde{M}$: RDBMS, we define the operator `op schema : -> Fun{rdbms}`, which can be used as follows: `t.schema(`$\widetilde{M}$`)`.

OclAny

The OclAny type is presented in the OCL specification as the supertype of all the types that appear in a MOF metamodel $\mathcal{M}$, except for OCL collection types. This type provides a set of operators that are inherited by all the types of a MOF metamodel. This type is not represented as a specific sort in our specification but its operators are defined in the `MODEL{OBJ :: TH-OBJECT}` theory as follows:

- *oclIsTypeOf:* This operator indicates if a value has a specific type, corresponding to the *isValueOf* relation. For example, the *oclIsTypeOf* operator is defined, in Maude notation, to check if a primitive type value is of sort `Bool` as follows:

  ```
      op _.‘oclIsTypeOf‘(‘Bool‘) : Bool -> Bool .
      eq B:Bool . oclIsTypeOf( Bool ) = true .
      op _.‘oclIsTypeOf‘(‘Bool‘) : String -> Bool .
      eq S:String . oclIsTypeOf( Bool ) = false .
      op _.‘oclIsTypeOf‘(‘Bool‘) : Int -> Bool .
      eq I:Int . oclIsTypeOf( Bool ) = false .
      op _.‘oclIsTypeOf‘(‘Bool‘) : Float -> Bool .
      eq F:Float . oclIsTypeOf( Bool ) = false .
      op _.‘oclIsTypeOf‘(‘Bool‘) : Enum -> Bool .
      eq E:Enum . oclIsTypeOf( Bool ) = false .
  ```

  This operator is likewise defined for the sorts `String`, `Int` and `Float`. Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the *oclIsTypeOf* operator is also defined for object type instances $\widetilde{o}$, such that $\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}}$. This operator checks whether an object $\widetilde{o}$ is a direct instance of a specific object type, without considering the specialization relation $<_s$ that may be defined between object type definitions in $\widetilde{\mathcal{M}}$. Given a specific object type name *CID*, such that $CID \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), Cid\#\mathcal{M}}$, the *oclIsTypeOf* operator is defined for object type instances as follows:

  ```
  var Obj:Object#OBJ .
  var CID:Cid#OBJ .

  op _.‘oclIsTypeOf‘(_‘) : Object#OBJ Cid#OBJ -> Bool .
  eq Obj . oclIsTypeOf( CID ) = (class(EObj) == CID) .
  ```

  In addition, the equations that indicate that an object is not a value of a data type are expressed, in Maude notation, as

  ```
  eq O:Object#OBJ . oclIsTypeOf( Bool ) = false .
  eq O:Object#OBJ . oclIsTypeOf( String ) = false .
  eq O:Object#OBJ . oclIsTypeOf( Int ) = false .
  eq O:Object#OBJ . oclIsTypeOf( Float ) = false .
  ```

- *oclIsKindOf:* This operator checks whether or not a type is valid for a specific value by considering subtype relationships. Among the data types in OCL, only the type Integer type is defined as a subtype of the Real type. As already mentioned, the sorts that correspond to these types, in MEL, are `Int` and `Float` respectively, which are not defined in the same kind. To provide the semantics of the operator that is described in the standard, we define the *oclIsKindOf* operator for the `Float` sort in the `MODEL{OBJ :: TH-OBJECT}` theory, in Maude notation, as follows:

  ```
  op _.‘oclIsKindOf‘(‘Float‘) : Bool -> Bool .
  eq B:Bool . oclIsKindOf( Float ) = false .
  op _.‘oclIsKindOf‘(‘Float‘) : String -> Bool .
  eq S:String . oclIsKindOf( Float ) = false .
  op _.‘oclIsKindOf‘(‘Float‘) : Int -> Bool .
  eq I:Int . oclIsKindOf( Float ) = true .
  op _.‘oclIsKindOf‘(‘Float‘) : Float -> Bool .
  eq F:Float . oclIsKindOf( Float ) = true .
  ```

  This operator is likewise defined for the sorts `Bool`, `String` and `Int`. Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the *oclIsKindOf* operator is also defined for object type instances $\widetilde{o}$, such that $\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), Object\#\mathcal{M}}$. This operator checks whether an object $\widetilde{o}$ is an instance of a specific object type, by considering the specialization relation $<_s$ that may be defined between object type definitions in $\widetilde{\mathcal{M}}$. Given a specific object type name *CID*, such that $CID \in T_{reflect_{\text{MOF}}(\widetilde{\mathcal{M}}), Cid\#\mathcal{M}}$, the *oclIsKindOf* operator is defined for object type instances as follows:

```
var Obj:Object#OBJ .
var CID:Cid#OBJ .

op _.‘oclIsKindOf‘(_‘) : Object#OBJ Cid#OBJ -> Bool .
eq Obj . oclIsKindOf( CID ) = (class(Obj) :: sortOf(CID)) .
```

where the function $sortOf : Cid\#\mathcal{M} \longrightarrow Sorts$ obtains the class sort that corresponds to a specific class constant by using the MEL reflective features[6].

- *oclAsType:* This partial operator provides support for casting or retyping an object into a (usually) more specific object type. The target type must be related to the type of the object that is being retyped by means of the specialization relation $<_s$ that is defined in a metamodel definition $\widetilde{\mathcal{M}}$ that provides the object types, i.e., one must be a subtype of the other. This operator is defined for object type instances as follows:

```
    var Obj:Object#OBJ .
    var CID:Cid#OBJ .

    op _.‘oclAsType‘(_‘) : Object#OBJ Cid#OBJ ~> Object#OBJ .
    ceq Obj . oclAsTypeOf( CID ) = Obj
    if Obj . oclIsKindOf( CID ) .
```

  Note that if the type name that is passed as argument is not a proper type for the object, the operator returns an undefined value. This operator is not defined for primitive type values nor enumeration literals in our algebraic specification.

OclType

This type introduces a metalevel feature in the OCL language that permits defining OCL types as instances of the *OclType* object type. However, a specific representation for this type is not provided in our algebraic specification. The dynamic type of a specific object can still be checked by means of the *oclIsTypeOf* and *oclIsKindOf* operators. The *OclType* type provides the operator *allInstances*, which is defined for all object types. Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ and an object type name *CID*, such that $CID \in T_{reflect_{\text{MOF}}(\widetilde{\mathcal{M}}), Cid\#\mathcal{M}}$, $CID.allInstances(\widetilde{M})$ obtains the set of instances of a specific object type in a given model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$. The operator *allInstances* is defined in the `MODEL{OBJ :: TH-OBJECT}` theory, in Maude notation, as follows:

```
var Obj : Object#OBJ .
var ObjCol : ObjectCollection{TH-OBJECT}{OBJ} .

op _.‘allInstances‘(_‘) : Cid#OBJ Configuration{TH-OBJECT}{OBJ}
    -> Set{TH-OBJECT}{OBJ} .
ceq CID . allInstances( << Obj ObjCol >> ) =
    (CID . allInstances( << ObjCol >>)) -> including( Obj )
if Obj . oclIsKindOf( CID ) .
eq CID . allInstances( << ObjColl >> ) = empty-set#OBJ [owise] .
```

where the operator

$$\_\texttt{->}including(\_) : Collection\{OBJ\} \times Object\#OBJ \longrightarrow Collection\{OBJ\}$$

permits adding an object to a collection of objects.

[6]In the algebraic specification, we use the MEL reflective features that are implemented in Maude to define the *oclIsKind* operator, instead of the *sortOf* operator. We have introduced this operator to make the definition of the operator easier to understand.

7.3 Algebraic Semantics of the *reflect* Function

The *reflect* function provides the algebraic semantics for the OCL constraints $\mathcal{C}$ that are meaningful for a specific metamodel $\mathcal{M}$ in a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. The $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory includes the theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$, which provides the algebraic semantics of the OCL predefined operators, by including the `OCL-COLLECTIONS{TH-OBJECT}{`$\mathcal{M}$`}` and `MODEL{`$\mathcal{M}$`}` theories. The $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory also provides a membership axiom of the form

$$\widetilde{M} : \mathcal{M} \;\wedge\; \mathit{condition}_1(\widetilde{M}) = \mathit{true} \;\wedge \cdots \wedge\; \mathit{condition}_n(\widetilde{M}) = \mathit{true} \Longrightarrow \widetilde{M} : (\mathcal{M}, \mathcal{C}),$$

where each constraint definition $\widetilde{c_i}$, in $\widetilde{\mathcal{C}} = \{\widetilde{c_1}, ..., \widetilde{c_n}\}$, corresponds to a boolean function $\mathit{condition}_i$ that is evaluated over a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, i.e., $\mathit{condition}_i(\widetilde{M})$. Therefore, when a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, satisfies all the constraints that are defined in $\widetilde{\mathcal{C}}$, $\widetilde{M}$ is considered a value of the constrained model type $[\![(\mathcal{M}, \mathcal{C})]\!]_{\mathrm{MOF}}$.

The OCL expression that constitutes an OCL constraint is a user-defined expression that may involve operators that are predefined for OCL types. The OCL expression that constitutes the OCL constraint that is taken as example is shown, by using the textual concrete syntax of the OCL language, as

```
[if] (self.column->[size]() [=] self.refersTo.column -> [size]()) [then]
  self.column->[forAll](c:Column |
    self.refersTo.column-> [at](self.column->[indexOf](c)).type
    [=] c.type
  )
[else]
  false
[endif]
```

where predefined operators are boxed. OCL constraints are usually given in textual format, which are parsed and expressed as model definitions $\widetilde{c}$, such that $\widetilde{c} : \mathrm{OCL}$ and $\mathit{root}(\widetilde{c}, \widetilde{\mathrm{OCL}}) : \textsc{Constraint}$. An OCL constraint $\widetilde{c}$ preserves a tree structure that is given by $\mathit{tree}(\widetilde{c}, \widetilde{\mathrm{OCL}})$.

An OCL constraint definition $\widetilde{c}$, such that $\widetilde{c} : \mathrm{OCL}$, $\mathit{root}(\widetilde{c}, \widetilde{\mathrm{OCL}}) : \textsc{OclConstraint}$, must be satisfied for all instances of the contextual object type that is referenced by $\textsc{Constraint}$ instance $\mathit{root}(\widetilde{c}, \widetilde{\mathrm{OCL}})$, which is obtained by means of the expression $\mathit{root}(\widetilde{c}, \widetilde{\mathrm{OCL}}).\mathit{constrainedElement}(\widetilde{\mathcal{M}})$. This semantics can be defined by using the OCL language itself. Assume that the textual representation of $\widetilde{c}$ is

```
context OT :
inv:  <boolean-body-expression>
```

The OCL constraint $\widetilde{c}$ can be defined by means of the OCL expression

```
OT.allInstances() -> forAll(self :  OT | < boolean-body-expression >).
```

The function $\mathit{transform} : [\![\mathrm{OCL}]\!]_{\mathrm{MOF}} \rightsquigarrow [\![\mathrm{OCL}]\!]_{\mathrm{MOF}}$ provides this transformation for a given OCL constraint definition $\widetilde{c}$.

The *reflect* function provides the semantics of OCL constraint definitions $\widetilde{c}$. On the one hand, the semantics of the predefined operators that may be referenced in an OCL constraint definition $\widetilde{c}$ is provided by the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory, as shown in Section 7.2. On the other hand, user defined expressions are defined in the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory by means of two functions:

- *getExpTheory:* The function

$$\begin{aligned}&\mathit{getExpTheory} :\\ &\quad [\![\textsc{OclExpression}]\!]_{\mathrm{MOF}} \times [\![\mathrm{OCL}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \times [\![\mathrm{OCL}_0]\!]_{\mathrm{MOF}}\\ &\quad \rightsquigarrow \mathit{SpecMEL}\end{aligned}$$

  provides the sorts and operators that are needed to define user-defined OCL expressions.

- *getExpTerm:* The function

$$\begin{array}{l} \mathit{getExpTerm} : \\ \quad [\![\textsc{OclExpression}]\!]_{\mathrm{MOF}} \times [\![\mathrm{OCL}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \times [\![\mathrm{OCL}_0]\!]_{\mathrm{MOF}} \\ \quad \rightsquigarrow \mathit{Terms} \end{array}$$

  represents a user-defined OCL expression as a term by using the predefined operators, which are provided by the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory, and the user-defined operators, which are provided by means of the *getExpTheory* function. This function maps a specific object $\widetilde{e}$, such that $\widetilde{e} : \textsc{OclExpression}$ and $\widetilde{e} \in \widetilde{c}$, to a term. The equational simplification of this term, by using the equations of the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory modulo the associativity and commutativity axioms, provides the evaluation of the corresponding OCL expression.

Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the *reflect* function is defined by the equalities

$$\begin{array}{l} \mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}) = \mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}) \ \cup \ (\varnothing,\ \varnothing,\ \varnothing,\ E_{\mathit{reflect}}) \ \cup \\ \displaystyle\bigcup_{\widetilde{c} \in \{\widetilde{c}:\mathrm{OCL} \mid \widetilde{c} \in \widetilde{\mathcal{C}}\}} \mathit{getExpTheory}(\mathit{root}(\mathit{transform}(\widetilde{c}), \widetilde{\mathrm{OCL}}),\ \mathit{transform}(\widetilde{c}), \widetilde{\mathcal{M}}, \mathit{empty\text{-}env}) \\ \text{when } \widetilde{\mathcal{C}} \neq \varnothing \end{array}$$

$$\begin{array}{l} \mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}) = \mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}) \ \cup \ (\varnothing,\ \varnothing,\ \varnothing,\ E_{\mathit{reflect}}{}') \\ \text{when } \widetilde{\mathcal{C}} = \varnothing, \end{array}$$

where

$$\begin{array}{l} E_{\mathit{reflect}} = \\ \quad \{(\widetilde{M} : \mathcal{M} \ \wedge \\ \quad \displaystyle\bigwedge_{\widetilde{c} \in \{\widetilde{c}:\mathrm{OCL} \mid \widetilde{c} \in \widetilde{\mathcal{C}}\}} \begin{array}{l}(\mathit{getExpTerm}(\mathit{root}(\mathit{transform}(\widetilde{c}), \widetilde{\mathrm{OCL}}), \\ \quad \mathit{transform}(\widetilde{c}), \widetilde{\mathcal{M}}, \mathit{empty\text{-}env}) = \mathit{true})\end{array} \\ \quad \Longrightarrow \widetilde{M} : (\mathcal{M}, \mathcal{C}) \\ \quad)\}, \end{array}$$

and

$$E_{\mathit{reflect}}{}' = \{(\widetilde{M} : \mathcal{M} \Longrightarrow \widetilde{M} : (\mathcal{M}, \mathcal{C}))\}.$$

The *getExpTheory* and *getExpTerm* functions traverse all the objects that constitute an OCL constraint $\widetilde{c}$ by means of the containment relation $<_c (\widetilde{c}, \widetilde{\mathrm{OCL}})$. In subsequent sections, we provide a detailed definition of both functions, indicating the subset of the OCL language that we have taken into account.

7.3.1 Preliminary concepts and functions

SpecMEL is the data type of *finitely-presented* MEL theories, that is, theories of the form $(S, <, \Omega, E \cup A)$, where all the components are finite. Without loss of generality we assume countable sets *Sorts*, *OpNames*, *VarNames*, *Ops*, and *ViewNames*, so that:

- each set of sorts S is a finite subset of *Sorts*;
- the operator names in Ω are a finite subset of *OpNames*;
- all variables appearing in $E \cup A$ belong to the set *Vars*, where

$$\mathit{Vars} = \{x : s \mid x \in \mathit{VarNames}, s \in \mathit{Sorts}\} \ \cup \ \{x : [s] \mid x \in \mathit{VarNames}, s \in \mathit{Sorts}\};$$

- *Ops* is the set of operators that can be defined in SpecMEL, which is defined by the equation

$$\mathit{Ops} = \{(f : s_1 \times \cdots \times s_n \to s) \mid f \in \mathit{OpNames} \ \ \wedge \ \ s, s_1, \ldots, s_n \in \mathit{Sorts}\};$$

- and *ViewNames* is the set of view names that can be used to instantiate a parameterized theory.

Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, we provide a set of functions that are used to define the *getExpTheory* and *getExpTerm* functions. We define some common domains to define this functions:

- Recall the notation $[\![\mathrm{OT}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ to define a subset of the cartesian product $[\![\mathrm{OT}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$, where OT is the sort of a specific object type, which is defined in the metamodel definition $\widetilde{\mathrm{MOF}}$. $(\widetilde{o}, \widetilde{\mathcal{M}}) \in [\![\mathrm{OT}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ iff $\widetilde{o} : \mathrm{OT}$, $\widetilde{\mathcal{M}} : \mathrm{MOF}$ and $\widetilde{o} \in \widetilde{\mathcal{M}}$.
- Given an object type OT, which is defined in the metamodel definition $\widetilde{\mathrm{OCL}}$, the domain $[\![\mathrm{OT}]\!]_{\mathrm{OclExp} \times \mathrm{OCL} \times \mathrm{MOF}}$ is defined as a subset of the cartesian product

$$[\![\mathrm{OT}]\!]_{\mathrm{MOF}} \times [\![\mathrm{OCL}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$$

  so that

$$\begin{gathered}(\widetilde{e}, \widetilde{c}, \widetilde{\mathcal{M}}) \in [\![\mathrm{OT}]\!]_{\mathrm{OclExp} \times \mathrm{OCL} \times \mathrm{MOF}} \\ \Leftrightarrow \\ \widetilde{e} : \mathrm{OT} \ \wedge \ \mathrm{OT} <_s^* \textsc{OclExpression} \ \wedge \\ \widetilde{c} : \mathrm{OCL} \ \wedge \ root(\widetilde{c}, \widetilde{\mathrm{OCL}}) : \textsc{IteratorExp} \ \wedge \ root(\widetilde{c}, \widetilde{\mathrm{OCL}}).name = \text{"}forAll\text{"} \ \wedge \\ \widetilde{e} \in \widetilde{c} \ \wedge \ \widetilde{\mathcal{M}} : \mathrm{MOF} \ \wedge \ root(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}}) : \textsc{Package} \ \wedge \\ root(\widetilde{c}, \widetilde{\mathrm{OCL}}).context(\widetilde{\mathcal{M}}).oclIsUndefined = false\end{gathered}$$

  where $<_s^*$ is the reflexive-transitive closure of the specialization relation $<_s$ that is defined in $\widetilde{\mathrm{OCL}}$.
- The domain *TypedVariables* is defined as a subset of the cartesian product

$$[\![\textsc{Variable}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}},$$

  where $(\widetilde{v}, \widetilde{\mathcal{M}}) \in$ *TypedVariables* iff $\widetilde{v} : \textsc{Variable}$, $\widetilde{\mathcal{M}} : \mathrm{MOF}$, and $\widetilde{v}.type(\widetilde{\mathcal{M}}).oclIsUndefined = false$.

Given a pair $(\widetilde{c}, \widetilde{\mathcal{M}})$, where $\widetilde{c} : \mathrm{OCL}$ and $\widetilde{\mathcal{M}} : \mathrm{MOF}$, the functions that are used to define the *getExpTheory* and *getExpTerm* functions are:

- *getBodyExpName:* An object $\widetilde{e}$, such that $\widetilde{e} : \textsc{LoopExp}$ and $\widetilde{e} \in \widetilde{c}$, defines an OCL expression in which a *loop* operator is referenced. In an object $\widetilde{e}$, $\widetilde{e}.body(\widetilde{c})$ refers to the root object of the OCL expression that constitutes the body of the operator. The function

$$getBodyExpName : \quad [\![\textsc{LoopExp}]\!]_{\mathrm{OclExp} \times \mathrm{OCL} \times \mathrm{MOF}} \longrightarrow OpNames$$

  maps a tuple $(\widetilde{e}, \widetilde{c}, \widetilde{\mathcal{M}})$, such that $(\widetilde{e}, \widetilde{c}, \widetilde{\mathcal{M}}) \in [\![\textsc{LoopExp}]\!]_{\mathrm{OclExp} \times \mathrm{OCL} \times \mathrm{MOF}}$, to an operator name, which constitutes a unique identifier for the body expression $\widetilde{e}.body(\widetilde{c})$ within the $reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory. $getBodyExpName(\widetilde{e}, \widetilde{c}, \widetilde{\mathcal{M}})$ represents the name of the body expression $\widetilde{e}.body(\widetilde{c})$. This function generates unique identifiers but this process is not detailed. In the examples, we will use intuitive names for the sake of understanding.
- *getViewName:* The body of the *iterate* operator is specified by means of the operator

```
    op _._‘(_;_;_‘) :
      Collection+{T} IterateBody{T} Variable Environment PreConfiguration{T}
        ~> Variable .
```

  This operator is defined in the `OCL-COLLECTIONS{T :: TRIV}` theory, as shown in Section 7.2. The sort of the operator name, `IterateBody{T}`, is qualified with the view, `T`, that is used to instantiate the `OCL-COLLECTIONS{T :: TRIV}` theory. This view name depends on the type of the elements of the source collection, to which the *iterate* operator is applied. This type can be obtained by querying the type of the *iterator* variable of the iterator operator. The function

$$getViewName : \quad TypedVariables \longrightarrow ViewNames$$

  maps a tuple $(\widetilde{e}, \widetilde{\mathcal{M}})$, such that $(\widetilde{e}, \widetilde{\mathcal{M}}) \in TypedVariables$, to a view name. Depending on the type of the source collection of the $\textsc{Variable}$ instance, the view name is generated as follows:

$$
\begin{aligned}
&getViewName(\widetilde{e},\ \widetilde{\mathcal{M}}) = \mathcal{M}\\
&\quad \text{when } \widetilde{e}.type(\widetilde{\mathcal{M}}) : \textsc{Class}\\
&getViewName(\widetilde{e},\ \widetilde{\mathcal{M}}) = \widetilde{e}.type(\widetilde{\mathcal{M}}).name\\
&\quad \text{when } \widetilde{e}.type(\widetilde{\mathcal{M}}) : \textsc{Enumeration}\\
&getViewName(\widetilde{e},\ \widetilde{\mathcal{M}}) = \texttt{Bool}\\
&\quad \text{when } \widetilde{e}.type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Boolean}}, \widetilde{\text{MOF}})\\
&getViewName(\widetilde{e},\ \widetilde{\mathcal{M}}) = \texttt{String}\\
&\quad \text{when } \widetilde{e}.type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{String}}, \widetilde{\text{MOF}})\\
&getViewName(\widetilde{e},\ \widetilde{\mathcal{M}}) = \texttt{Int}\\
&\quad \text{when } \widetilde{e}.type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Integer}}, \widetilde{\text{MOF}})\\
&getViewName(\widetilde{e},\ \widetilde{\mathcal{M}}) = \texttt{Float}\\
&\quad \text{when } \widetilde{e}.type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Real}}, \widetilde{\text{MOF}})
\end{aligned}
$$

- *getBodyExpSort:* The body of an iterator operator, such as *forAll*, is defined by means of the operator

```
op _._`(_;_`) :
  Collection+{T} BodyExpSort{T} Variable Environment Configuration{T}
    ~> Variable .
```

where the sort `BodyExpSort{T}` depends on both the type of the source collection, to which the iterator operator is applied and the type of the value that is returned by the iterator operator. The function

$$getBodyExpSort : \quad [\![\textsc{IteratorExp}]\!]_{\textsc{OclExp}\times\text{OCL}\times\text{MOF}} \longrightarrow Sorts$$

generates the corresponding `BodyExpSort{T}` sort depending on both the type of the source collection of an IteratorExp instance and the type of the returning value, as follows:

$$
\begin{aligned}
&getBodyExpSort(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{Body}\{\boxed{getViewName(\widetilde{e}.iterator(\widetilde{c}),\ \widetilde{\mathcal{M}})}\}\\
&\quad \text{when } \widetilde{e}.body(\widetilde{c}).type(\widetilde{\mathcal{M}}) : \textsc{Class}\\
&getBodyExpSort(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{EnumBody}\{\boxed{getViewName(\widetilde{e}.iterator(\widetilde{c}),\ \widetilde{\mathcal{M}})}\}\\
&\quad \text{when } \widetilde{e}.body(\widetilde{c}).type(\widetilde{\mathcal{M}}) : \textsc{Enumeration}\\
&getBodyExpSort(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{BoolBody}\{\boxed{getViewName(\widetilde{e}.iterator(\widetilde{c}),\ \widetilde{\mathcal{M}})}\}\\
&\quad \text{when } \widetilde{e}.body(\widetilde{c}).type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Boolean}}, \widetilde{\text{MOF}})\\
&getBodyExpSort(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{StringBody}\{\boxed{getViewName(\widetilde{e}.iterator(\widetilde{c}),\ \widetilde{\mathcal{M}})}\}\\
&\quad \text{when } \widetilde{e}.body(\widetilde{c}).type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{String}}, \widetilde{\text{MOF}})\\
&getBodyExpSort(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{IntBody}\{\boxed{getViewName(\widetilde{e}.iterator(\widetilde{c}),\ \widetilde{\mathcal{M}})}\}\\
&\quad \text{when } \widetilde{e}.body(\widetilde{c}).type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Integer}}, \widetilde{\text{MOF}})\\
&getBodyExpSort(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{FloatBody}\{\boxed{getViewName(\widetilde{e}.iterator(\widetilde{c}),\ \widetilde{\mathcal{M}})}\}\\
&\quad \text{when } \widetilde{e}.body(\widetilde{c}).type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Real}}, \widetilde{\text{MOF}})
\end{aligned}
$$

Note that the type of the elements of the source collection coincides with the type of the *iterator* variable.

- *getPreConfigurationSort:* In the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory, when the *source* collection of a *loop* operator is a collection of objects, the body expression of the *loop* operator may need the metamodel definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, to enable the navigation through object-typed properties. The function

$$getPreConfigurationSort : \quad [\![\textsc{LoopExp}]\!]_{\textsc{OclExp} \times \text{OCL} \times \text{MOF}} \longrightarrow Vars$$

permits obtaining a variable for the source collection depending on its type as follows:

$$\begin{aligned} &getPreConfigurationSort(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{model:ModelType}\{\mathcal{M}\} \\ &\quad \text{when } \widetilde{e}.iterator(\widetilde{c}).type(\widetilde{\mathcal{M}}) : \textsc{Class} \\ &ModelType(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{pc:PreConfiguration}\{ \\ &\qquad\qquad \boxed{getViewName(\widetilde{e}.iterator(\widetilde{c}),\ \widetilde{\mathcal{M}})} \\ &\qquad \} \\ &\quad \text{otherwise} \end{aligned}$$

- *getSortedVariableValue:* Given a variable of the form (*FooVar* = *BarValue*), where *FooVar : VariableName* and *BarValue* $\in \mathcal{D}$, where $\mathcal{D}$ represents the set of all definable values in the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory. Variables of this kind are used to define the environment of an OCL expression in the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory. To obtain the field *BarValue* of the previous variable, we use a typed variable, which belongs to the set *Vars*. Given a pair $(\widetilde{e}, \widetilde{\mathcal{M}})$, such that $(\widetilde{e}, \widetilde{\mathcal{M}}) \in TypedVariables$ and $(\widetilde{e}, \widetilde{c}, \widetilde{\mathcal{M}}) \in [\![\textsc{Variable}]\!]_{\textsc{OclExp} \times \text{OCL} \times \text{MOF}}$, the function

$$getSortedVariableValue : TypedVariables \longrightarrow Vars$$

obtains the corresponding variable as follows:

$$\begin{aligned} &getSortedVariableValue(\widetilde{e},\ \widetilde{\mathcal{M}}) = \boxed{\widetilde{e}.name}\texttt{Value:Collection+}\{ \\ &\qquad \boxed{getViewName(\widetilde{e}.iterator(\widetilde{c}),\ \widetilde{\mathcal{M}})} \\ &\quad \} \\ &\quad \text{where } \widetilde{e} : \textsc{Variable} \end{aligned}$$

- *getParameter :* A variable of the form (*FooVar* = *BarValue*), such that *FooVar : VariableName* and *BarValue* $\in \mathcal{D}$, can be used to define the environment of an OCL expression in the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory. Given a pair $(\widetilde{e}, \widetilde{\mathcal{M}})$, such that $(\widetilde{e}, \widetilde{\mathcal{M}}) \in TypedVariables$ and $(\widetilde{e}, \widetilde{c}, \widetilde{\mathcal{M}}) \in [\![\textsc{Variable}]\!]_{\textsc{OclExp} \times \text{OCL} \times \text{MOF}}$, the function

$$getParameter : TypedVariables \longrightarrow Vars$$

provides the OCL variable that corresponds to a Variable instance as follows:

$$\begin{aligned} &getParameter(\widetilde{e},\ \widetilde{\mathcal{M}}) = \boxed{\widetilde{e}.name}\ \texttt{=}\ \boxed{getSortedVariableValue(\widetilde{e},\ \widetilde{\mathcal{M}})} \\ &\quad \text{where } \widetilde{e} : \textsc{Variable} \end{aligned}$$

For example, given a Variable instance $\widetilde{v}$ such that $\widetilde{v}.name$ = "*intVar*" and $\widetilde{v}.type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Integer}}, \widetilde{\text{MOF}})$, $getParameter(\widetilde{e}, \widetilde{\mathcal{M}})$ results in the expression: `intVar = intVarValue:Collection+{Int}`.

- *getEnvironment:* The function

$$getEnvironment : [\![\text{OCL}_0]\!]_{\text{MOF}} \times [\![\text{MOF}]\!]_{\text{MOF}} \rightsquigarrow Terms$$

is defined for pairs of the form $(EnvVars, \widetilde{\mathcal{M}})$, where *EnvVars*, such that $EnvVars : \text{OCL}_0$, represents a set of Variable instances, $\widetilde{\mathcal{M}}$ is the metamodel definition of the metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, and $\forall \widetilde{v}(\widetilde{v} : \textsc{Variable} \ \wedge\ \widetilde{v} \in EnvVars \rightarrow \widetilde{v}.type(\widetilde{\mathcal{M}}) \neq \varnothing)$. The *getEnvironment* function defines a term of sort `Environment` from the set *EnvVars* of variables, which represents the environment of an OCL expression. The *getEnvironment* function is defined as follows:

$$
\begin{aligned}
&\mathit{getEnvironment}(\mathit{EnvVars},\ \widetilde{\mathcal{M}}) = \\
&\qquad \bigcup_{\widetilde{v}\,\in\,\{\widetilde{v}:\textsc{VariableExp}\ \mid\ \widetilde{v}\,\in\,\mathit{EnvVars}\}} \{\mathit{getParameter}(\widetilde{v},\ \widetilde{\mathcal{M}})\} \\
&\text{when } \mathit{EnvVars} \neq \emptyset \\
&\mathit{getEnvironment}(\mathit{EnvVars},\ \widetilde{\mathcal{M}}) = \texttt{empty-env} \\
&\text{when } \mathit{EnvVars} = \emptyset
\end{aligned}
$$

- *getPropertyCallSort:* Given an object $\widetilde{o}$ = `< OID : CID | name : "Foo", PS>`, such that $\widetilde{o} \in \widetilde{M}$ and $\widetilde{M} : \mathcal{M}$, the field `value` of the `prop` property is obtained by means of the expression `o.prop`, where `prop` is defined as a constant `prop : StringFun{`$\mathcal{M}$`}` for this example. Properties of this kind are defined in $\widetilde{\mathcal{M}}$ by means of a Property instance $\widetilde{p}$. Given a pair $(\widetilde{p}, \widetilde{\mathcal{M}})$, such that $(\widetilde{p}, \widetilde{\mathcal{M}}) \in [\![\textsc{Property}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$, the function

$$\mathit{getPropertyCallSort} : [\![\textsc{Property}]\!]_{\mathrm{MOF}} \otimes [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow \mathit{Sorts}$$

  obtains the sort of the constant that constitutes the property projector name, depending on the type of the corresponding Property instance $\widetilde{p}$, as follows:

$$
\begin{aligned}
&\mathit{getPropertyCallSort}(\widetilde{p},\ \widetilde{\mathcal{M}}) = \texttt{Fun}\{\mathcal{M}\} \\
&\quad \text{when } \widetilde{p}.\mathit{type}(\widetilde{\mathcal{M}}) : \textsc{Class} \\
&\mathit{getPropertyCallSort}(\widetilde{p},\ \widetilde{\mathcal{M}}) = \texttt{EnumFun}\{\mathcal{M}\} \\
&\quad \text{when } \widetilde{p}.\mathit{type}(\widetilde{\mathcal{M}}) : \textsc{Enumeration} \\
&\mathit{getPropertyCallSort}(\widetilde{p},\ \widetilde{\mathcal{M}}) = \texttt{BoolFun}\{\mathcal{M}\} \\
&\quad \text{when } \widetilde{p}.\mathit{type}(\widetilde{\mathcal{M}}) = \mathit{root}(\widetilde{\textsc{Boolean}}, \widetilde{\mathrm{MOF}}) \\
&\mathit{getPropertyCallSort}(\widetilde{p},\ \widetilde{\mathcal{M}}) = \texttt{StringFun}\{\mathcal{M}\} \\
&\quad \text{when } \widetilde{p}.\mathit{type}(\widetilde{\mathcal{M}}) = \mathit{root}(\widetilde{\textsc{String}}, \widetilde{\mathrm{MOF}}) \\
&\mathit{getPropertyCallSort}(\widetilde{p},\ \widetilde{\mathcal{M}}) = \texttt{IntFun}\{\mathcal{M}\} \\
&\quad \text{when } \widetilde{p}.\mathit{type}(\widetilde{\mathcal{M}}) = \mathit{root}(\widetilde{\textsc{Integer}}, \widetilde{\mathrm{MOF}}) \\
&\mathit{getPropertyCallSort}(\widetilde{p},\ \widetilde{\mathcal{M}}) = \texttt{FloatFun}\{\mathcal{M}\} \\
&\quad \text{when } \widetilde{p}.\mathit{type}(\widetilde{\mathcal{M}}) = \mathit{root}(\widetilde{\textsc{Real}}, \widetilde{\mathrm{MOF}})
\end{aligned}
$$

7.3.2 User-Defined OCL Type Operators: *getExpTheory*

The function

$$
\begin{aligned}
&\mathit{getExpTheory} : \\
&\quad [\![\textsc{OclExpression}]\!]_{\mathrm{MOF}} \times [\![\mathrm{OCL}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \times [\![\mathrm{OCL}_0]\!]_{\mathrm{MOF}} \\
&\quad \rightsquigarrow \mathit{SpecMEL}
\end{aligned}
$$

provides the sorts and operators that are needed to define user-defined OCL expressions. This function is defined for tuples of the form $(\widetilde{e}, \widetilde{c}, \widetilde{\mathcal{M}}, \mathit{EnvVars})$, where:

$$
\begin{aligned}
&(\widetilde{e}, \widetilde{c}, \widetilde{\mathcal{M}}) \in [\![\textsc{OclExpression}]\!]_{\mathrm{OclExp}\times\mathrm{OCL}\times\mathrm{MOF}}, \\
&\mathit{EnvVars} : \mathrm{OCL}_0, \mathit{and} \\
&\forall \widetilde{v}(\widetilde{v} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{OCL}}), \mathit{Object\#OCL}} \ \wedge\ \widetilde{v} \in \mathit{EnvVars} \rightarrow \widetilde{v} : \textsc{Variable}).
\end{aligned}
$$

The semantics of the *getExpTheory* function is provided by means of mappings that project the OclExpression instances that constitute an OCL constraint definition $\widetilde{c}$ to sorts and operators in the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$

theory. The *getExpTheory* function traverses all the OclExpression instances that constitute $\widetilde{c}$ by considering the containment relation that is defined by $<_c (\widetilde{c}, \widetilde{\mathrm{OCL}})$, following a top-down strategy. The *getExpTheory* function is defined for the following object types of the OCL metamodel: IfExp, IterateExp, IteratorExp, LetExp, LiteralExp, OperationCallExp, PropertyCallExp, VariableExp, and TypeExp; as described in subsequent paragraphs.

IfExp

An IfExp instance defines an OCL expression that results in one of two alternative expressions, *thenExpression* and *elseExpression*, depending on the evaluated value of a condition. No operators are generated for an IfExp instance by means of the *getExpTheory* function. Instead its contained *condition*, *thenExpression* and *elseExpression* OclExpression instances are processed:

$$\begin{array}{l} getExpTheory(\widetilde{e}, \widetilde{c}, \widetilde{\mathcal{M}}, EnvVars) = \\ \quad getExpTheory(\widetilde{e}.condition(\widetilde{c}), \widetilde{c}, \widetilde{\mathcal{M}}, EnvVars) \cup \\ \quad getExpTheory(\widetilde{e}.thenExpression(\widetilde{c}), \widetilde{c}, \widetilde{\mathcal{M}}, EnvVars) \cup \\ \quad getExpTheory(\widetilde{e}.elseExpression(\widetilde{c}), \widetilde{c}, \widetilde{\mathcal{M}}, EnvVars) \\ where\ \widetilde{e} : \textsc{IfExp} \end{array}$$

IterateExp

An IterateExp instance represents an expression which evaluates its *body* expression for each element of a *source* collection. It acts as a loop construct that iterates over the elements of its source collection and results in a value. The evaluated value of the body expression in each iteration step becomes the new value for the *result* variable for the next iteration step. The result can be of any type and is defined by the *result* property. The *body* expression, and the *iterator* and *result* variables are defined by the user. An IteratorExp instance adds the *iterator* and *result* variables to the environment of the OCL expression, so that inner expressions in the tree $tree(\widetilde{c}, \widetilde{\mathrm{OCL}})$ can refer to them: the iterate body expression. An IterateExp instance is represented in the $reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory by means of the mapping:

$$\begin{array}{l} getExpTheory(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad getExpTheory(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) \cup \\ \quad getExpTheory(\widetilde{e}.body(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars \cup \ll \widetilde{e}.iterator(\widetilde{c})\ \ \widetilde{e}.result(\widetilde{c}) \gg) \cup \\ \quad (\varnothing,\ \varnothing,\ \Omega_{IterateExp},\ E_{IterateExp}) \\ where\ \widetilde{e} : \textsc{IterateExp} \end{array}$$

where

$$\begin{array}{l} \Omega_{IterateExp} = \{ \\ \quad (\texttt{op}\ \boxed{\widetilde{e}.iterator(\widetilde{c}).name}\ \ \texttt{:}\ \ \texttt{->}\ \texttt{VariableName .}), \\ \quad (\texttt{op}\ \boxed{\widetilde{e}.result(\widetilde{c}).name}\ \ \texttt{:}\ \ \texttt{->}\ \texttt{VariableName .}), \\ \quad (\texttt{op}\ \boxed{getBodyExpName(\widetilde{e}.body(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}})}\ \ \texttt{:}\ \ \texttt{->} \\ \quad\quad \texttt{IterateBody\{}\boxed{getViewName(\widetilde{e}.iterator(\widetilde{c}),\ \widetilde{\mathcal{M}})}\texttt{\}}\ \ \texttt{.}) \\ \} \end{array}$$

and

$$
\begin{array}{l}
E_{IterateExp} = \{ \\
\quad (\\
\qquad \texttt{eq}\ \boxed{getSortedVariableValue(\widetilde{e}.iterator(\widetilde{c}),\ \widetilde{\mathcal{M}})}\ \ \texttt{.} \\
\qquad \boxed{getBodyExpName(\widetilde{e}.body(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}})} \\
\qquad \texttt{(} \\
\qquad \boxed{getParameter(\widetilde{e}.result(\widetilde{c}),\ \widetilde{\mathcal{M}})}\ \ \texttt{;} \\
\qquad \boxed{getEnvironment(EnvVars,\ \widetilde{\mathcal{M}})}\ \ \texttt{;} \\
\qquad \boxed{getPreConfigurationSort(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}})} \\
\qquad \texttt{) =} \\
\qquad \texttt{(} \\
\qquad \boxed{(\widetilde{e}.result(\widetilde{c}).name)} \\
\qquad \texttt{=} \\
\qquad \boxed{(getExpTerm(\widetilde{e}.body(\widetilde{c}),\ \widetilde{c},\ EnvVars \cup \ll \widetilde{e}.iterator(\widetilde{c})\ \ \widetilde{e}.result(\widetilde{c}) \gg))} \\
\qquad \texttt{) .} \\
\quad) \\
\}
\end{array}
$$

For example, the OCL expression that sums the integer values in a collection of integers is defined, in OCL textual concrete syntax, as

```
Set{1, 2, 3} -> iterate( i:  Integer, sum:  Integer = 0 | sum + i ).
```

The *body* expression of the `IterateExp` that represents this expression is represented in the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory, in Maude notation, as

```
op i : -> VariableName .
op sum : -> VariableName .
op integerSum : -> iterateBody{Int} .

eq iValue:Collection+{Int} . integerSum(
    sum = sumValue:Collection+{Int} ;
    empty-env ;
    pc:PreConfiguration{Int}
) = ...
```

IteratorExp

An IteratorExp instance represents an expression that evaluates its *body* expression for each element of a *source* collection. It acts as a loop construct that iterates over the elements of its source collection and results in a value. The type of the iterator expression depends on the name of the expression, and sometimes on the type of the associated source expression. An IterateExp instance adds the *iterator* variable to the environment of the OCL expression, so that this variable can be referenced in its body expression. The operators that are generated for the *body* expression of an IterateExp instance in the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory are given as follows:

$$\begin{aligned}
&getExpTheory(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
&\quad getExpTheory(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) \cup \\
&\quad getExpTheory(\widetilde{e}.body(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars \cup \ll \widetilde{e}.iterator(\widetilde{c}) \gg) \cup \\
&\quad (\varnothing,\ \varnothing,\ \Omega_{IterateExp},\ E_{IterateExp}) \\
&when\ \widetilde{e} : \textsc{IteratorExp}\ \wedge\ \widetilde{e}.body(\widetilde{c}).type(\widetilde{\mathcal{M}}) \neq root(\widetilde{\textsc{Boolean}}, \widetilde{\mathrm{MOF}})
\end{aligned}$$

$$\begin{aligned}
&getExpTheory(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
&\quad getExpTheory(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) \cup \\
&\quad getExpTheory(\widetilde{e}.body(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars \cup \ll \widetilde{e}.iterator(\widetilde{c}) \gg) \cup \\
&\quad (\varnothing,\ \varnothing,\ \Omega_{IterateExp},\ E_{IterateExp}{}') \\
&when\ \widetilde{e} : \textsc{IteratorExp}\ \wedge\ \widetilde{e}.body(\widetilde{c}).type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Boolean}}, \widetilde{\mathrm{MOF}})
\end{aligned}$$

where

$$\begin{aligned}
&\Omega_{IteratorExp} = \{ \\
&\quad (\texttt{op}\ \boxed{\widetilde{e}.iterator(\widetilde{c}).name}\ \texttt{:\ \ ->\ VariableName\ .}), \\
&\quad (\texttt{op}\ \boxed{getBodyExpName(\widetilde{e}.body(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}})}\ \texttt{:\ \ ->}\ \boxed{getBodyExpSort(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}})}\ \texttt{.}) \\
&\}
\end{aligned}$$

and

$$\begin{aligned}
&E_{IteratorExp} = \{ \\
&\quad (\\
&\qquad \texttt{eq}\ \boxed{getSortedVariableValue(\widetilde{e}.iterator(\widetilde{c}),\ \widetilde{\mathcal{M}})}\ \texttt{.} \\
&\qquad\ \ \boxed{getBodyExpName(\widetilde{e}.body(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}})} \\
&\qquad \texttt{(} \\
&\qquad\ \ \boxed{getEnvironment(EnvVars,\ \widetilde{\mathcal{M}})}\ \texttt{;} \\
&\qquad\ \ \boxed{getPreConfigurationSort(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}})} \\
&\qquad \texttt{) =} \\
&\qquad\ \ \boxed{(getExpTerm(\widetilde{e}.body(\widetilde{c}),\ \widetilde{c},\ EnvVars \cup \ll \widetilde{e}.iterator(\widetilde{c}) \gg))}\ \texttt{.} \\
&\quad) \\
&\}
\end{aligned}$$

```
$E_{IteratorExp}{}' = \{$
  (
    eq $\boxed{getSortedVariableValue(\widetilde{e}.iterator(\widetilde{c}),\ \widetilde{\mathcal{M}})}$ .
    $\boxed{getBodyExpName(\widetilde{e}.body(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}})}$
    (
    $\boxed{getEnvironment(EnvVars,\ \widetilde{\mathcal{M}})}$ ;
    $\boxed{getPreConfigurationSort(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}})}$
    ) =
    $\boxed{(getExpTerm(\widetilde{e}.body(\widetilde{c}),\ \widetilde{c},\ EnvVars \cup \ll \widetilde{e}.iterator(\widetilde{c}) \gg))}$ .
  ),
  (
    eq $\boxed{getSortedVariableValue(\widetilde{e}.iterator(\widetilde{c}),\ \widetilde{\mathcal{M}})}$ .
    $\boxed{getBodyExpName(\widetilde{e}.body(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}})}$
    (E:Environment ; $\boxed{getPreConfigurationSort(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}})}$) = false [owise] .
  )
$\}$
```

For example, we consider the metamodel definition that is shown in Fig. 7.7. The following OCL expression checks that the value of the `a` property of any A instance in a model definition $\widetilde{M}$, such that $\widetilde{M}$: $\widetilde{\text{EXAMPLE}}$, is odd:

```
A.allInstances() -> forAll(objA : A | objA.a.mod(2) <> 0).
```

The operator that is needed to represent the body expression of the `forAll` operator is represented, in Maude notation, as follows:

```
op isOdd : -> BoolBody{Example} .

eq objA:Collection+{Example} . isOdd (
    empty-env ;
    model:ModelType{Example}
) = ...
eq objA:Collection+{Example} . isOdd (
    E:Environment;
    model:ModelType{Example}
) = false [owise] .
```

LetExp

A LETEXP instance represents a special expression that defines a new variable with an initial value. A variable defined by a LETEXP instance cannot change its value, which represents the evaluated value of an *initial expression.* The variable is visible in the *in* expression. No operators are needed to represent a LETEXP instance in the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory. In our approach, the variable that is declared in a *Let* expression is not added to the environment of the expression $\widetilde{e}.in(\widetilde{\mathcal{M}})$. Therefore, the *getExpTerm* function obtains the term that represents the value of a *let* variable, $\widetilde{e}.variable(\widetilde{\mathcal{M}})$, by processing the initial expression that is associated to the *let* variable, i.e., $\widetilde{e}.variable(\widetilde{\mathcal{M}}).initExpression(\widetilde{\mathcal{M}})$, as show in the following section. The equation that defines the *getExpTheory* function for the LETEXP object type is

$$\begin{array}{l} getExpTheory(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad getExpTheory(\widetilde{e}.in(\widetilde{\mathcal{M}}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) \\ where\ \widetilde{e} : \text{LetExp} \end{array}$$

LiteralExp

A LiteralExp instance represents an expression with no arguments producing a value. No operators are needed to represent a LiteralExp instance in the *reflect*$(\widetilde{\mathcal{M}},\widetilde{\mathcal{C}})$ theory.

$$\begin{array}{l} getExpTheory(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = (\varnothing,\ \varnothing,\ \varnothing,\ \varnothing) \\ where\ \widetilde{e} : \text{LiteralExp} \end{array}$$

OperationCallExp

An OperationCallExp instance refers to an operation, which is either predefined for the OCL types or for an object type in $\widetilde{\mathcal{M}}$. These operations are defined as data in the metamodel definition $\widetilde{\text{OclStdLib}}$, and we have provided their algebraic semantics in Section 7.2. A OperationCallExp instance may contain a list of *argument* expressions if the operation is defined to have parameters. No operators are generated for an OperationCallExp instance. Instead, the *getExpTheory* processes the *argument* expressions that are contained in it as follows:

$$\begin{array}{l} getExpTheory(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \bigcup_{\widetilde{arg}\ \in\ \{\widetilde{arg}:\text{OclExpression}\ \mid\ \widetilde{arg}\ \in\ \widetilde{e}.argument(\widetilde{c})\}} getExpTheory(\widetilde{arg},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) \\ when\ \widetilde{e} : \text{OperationCallExp} \end{array}$$

PropertyCallExp

A PropertyCallExpression instance is a reference to a Property instance that is defined in the metamodel definition $\widetilde{\mathcal{M}}$. The operators that are needed to project the value of an object property in an OCL expression are defined by means of the *getExpTheory* as follows:

$$\begin{array}{l} getExpTheory(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad (\varnothing,\ \varnothing,\ \Omega_{PropertyCallExp},\ E_{PropertyCallExp}) \\ when\ \widetilde{e} : \text{LetExp}\ \wedge\ \widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).type(\widetilde{\mathcal{M}}) : \text{DataType} \end{array}$$

$$\begin{array}{l} getExpTheory(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad (\varnothing,\ \varnothing,\ \Omega_{PropertyCallExp},\ E_{PropertyCallExp}{}') \\ when\ \widetilde{e} : \text{LetExp}\ \wedge\ \widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).type(\widetilde{\mathcal{M}}) : \text{Class} \end{array}$$

where

$$\begin{array}{l} \Omega_{PropertyCallExp} = \{ \\ \quad (\texttt{op}\ \boxed{\widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).name}\ \ \texttt{:}\ \ \texttt{->} \\ \qquad \boxed{getPropertyCallSort(\widetilde{e}.referredProperty(\widetilde{\mathcal{M}}),\widetilde{\mathcal{M}})}\ \ \texttt{.)} \\ \} \end{array}$$

and

$$\begin{array}{l}
E_{PropertyCallExp} = \{ \\
\quad (\\
\qquad \texttt{eq < OID:Oid : CID:}\boxed{\widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).class(\widetilde{\mathcal{M}}).name}\ \texttt{|} \\
\qquad\quad \texttt{(}\boxed{\widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).name}\ \texttt{: Value:Collection+\{}\boxed{getViewName(\widetilde{e},\widetilde{\mathcal{M}})}\texttt{\}),} \\
\qquad\quad \texttt{PS:PropertySet\#}\mathcal{M}\texttt{> .}\ \boxed{referredProperty(\widetilde{\mathcal{M}}).name}\ \texttt{=} \\
\qquad\qquad \texttt{Value:Collection+\{}\boxed{getViewName(\widetilde{e},\widetilde{\mathcal{M}})}\texttt{\} .} \\
\quad), \\
\quad (\\
\qquad \texttt{< OID:Oid : CID:}\boxed{\widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).class(\widetilde{\mathcal{M}}).name}\ \texttt{|} \\
\qquad\quad \texttt{(}\boxed{\widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).name}\texttt{), PS:PropertySet\#}\mathcal{M}\texttt{> .} \\
\qquad\quad \boxed{referredProperty(\widetilde{\mathcal{M}}).name}\ \texttt{=} \\
\qquad\qquad \boxed{defaultValueProperty(\widetilde{e}.referredProperty(\widetilde{\mathcal{M}}),\widetilde{\mathcal{M}})}\ \texttt{.} \\
\quad) \\
\}
\end{array}$$

$$\begin{array}{l}
{E_{PropertyCallExp}}' = \{ \\
\quad (\\
\qquad \texttt{eq < OID:Oid : CID:}\boxed{\widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).class(\widetilde{\mathcal{M}}).name}\ \texttt{|} \\
\qquad\quad \texttt{(}\ \boxed{\widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).name}\ \texttt{: Value:Collection+\{Oid\}),} \\
\qquad\quad \texttt{PS:PropertySet\#}\mathcal{M}\texttt{> .}\ \boxed{referredProperty(\widetilde{\mathcal{M}}).name}\ \texttt{=} \\
\qquad\qquad \texttt{Value:Collection+\{Oid\} .} \\
\quad), \\
\quad (\\
\qquad \texttt{eq < OID:Oid : CID:}\boxed{\widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).class(\widetilde{\mathcal{M}}).name}\ \texttt{|} \\
\qquad\quad \texttt{(}\ \boxed{\widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).name}\ \texttt{: Value:Collection+\{Oid\}),} \\
\qquad\quad \texttt{PS:PropertySet\#}\mathcal{M}\texttt{> .}\boxed{referredProperty(\widetilde{\mathcal{M}}).name}\texttt{(model:ModelType\{}\mathcal{M}\texttt{\}) =} \\
\qquad\qquad \texttt{search(Value:Collection+\{Oid\}, model:ModelType\{}\mathcal{M}\texttt{\}) .} \\
\quad) \\
\}
\end{array}$$

where the `defaultValueProperty` function obtains the default value that is defined for an object property taking into account its meta-property values, and the `search` function projects objects of a model definition given a specific collection of object identifiers. In the metamodel definition $\widetilde{\mathrm{RDBMS}}$, shown in Fig. 6.7, to query the `name` value-typed property of the RModelElement object type, the following operators are defined in the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory:

```
op name : -> StringFun{rdbms} .
```

```
eq < OID:Oid : CID:RModelElement |
    name : Value:Collection+{String}, PS:PropertySet#rdbms >
    . name =
    Value:Collection+{String} .
eq < OID:Oid : CID:RModelElement | name, PS:PropertySet#rdbms >
    . name = "" .
```

On the other hand, to query the `schema` object-typed property of the Table object type, the following operators are defined in the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory:

```
op schema : -> Fun{rdbms} .

eq < OID:Oid : CID:Table |
    name : Value:Collection+{Oid}, PS:PropertySet#rdbms >
    . schema =
    Value:Collection+{Oid} .
eq < OID:Oid : CID:RModelElement |
    name : Value:Collection+{Oid}, PS:PropertySet#rdbms >
    . schema( model:ModelType{rdbms} ) =
    search(Value:Collection+{Oid}, model:ModelType{rdbms}) .
```

VariableExp

A VariableExp instance represents an expression that consists of a reference to a variable. References to the variables *self*, the *result* variable of an iterate expression, or variables that are defined in *Let* expressions are examples of such variable expressions. No operators are defined in this case.

$$\begin{array}{ll} \mathit{getExpTheory}(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{EnvVars}) = & (\emptyset,\ \emptyset,\ \emptyset,\ \emptyset) \\ \mathit{when}\ \widetilde{e} : \textsc{VariableExp} & \end{array}$$

TypeExp

A TypeExp is an expression used to refer to an existing meta type within an expression. In this case, neither sorts nor operators are generated.

$$\begin{array}{ll} \mathit{getExpTheory}(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{EnvVars}) = & (\emptyset,\ \emptyset,\ \emptyset,\ \emptyset) \\ \mathit{when}\ \widetilde{e} : \textsc{TypeExp} & \end{array}$$

7.3.3 Algebraic Semantics of OCL Expressions: *getExpTerm*

We define the set *Terms* as the set of terms that can be defined by means of any mel theory in *SpecMEL*. The function

$$\begin{array}{ll} \mathit{getExpTerm}: & [\![\textsc{OclExpression}]\!]_{\mathrm{MOF}} \times [\![\mathrm{OCL}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \times [\![\mathrm{OCL}_0]\!]_{\mathrm{MOF}} \\ & \rightsquigarrow \mathit{Terms} \end{array}$$

represents an OCL expression as a term by using the predefined operators for OCL types, which are specified in the `OCL-COLLECTIONS{T :: TRIV}` and `MODEL{OBJ :: TH-OBJECT}` theories, and user-defined operators, which are provided by means of the *getExpTheory* function from a specific OCL expression. The operators that are defined in the signature of the resulting *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory are provided in mixfix notation and can be regarded as the context-free grammar of the OCL language. Terms that are built using such operators can be regarded as OCL expressions, whose semantics is provided by means of equations. An OCL expression is evaluated by reducing a term of this type using the equations of the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory modulo associativity and commutativity. The canonical form of a term that represents an OCL expression constitutes the resulting value of the OCL expression.

The *getExpTerm* function is defined for tuples of the form $(\widetilde{e}, \widetilde{c}, \widetilde{\mathcal{M}}, EnvVars)$, where:

$$(\widetilde{e}, \widetilde{c}, \widetilde{\mathcal{M}}) \in [\![\textsc{OclExpression}]\!]_{\textsc{OclExp} \times \textsc{OCL} \times \textsc{MOF}},$$
$$EnvVars : \text{OCL}_0, \textit{and}$$
$$\forall \widetilde{v}(\widetilde{v} \in T_{reflect_{\text{MOF}}(\widetilde{\text{OCL}}), Object\#OCL} \wedge \widetilde{v} \in EnvVars \rightarrow \widetilde{v} : \textsc{Variable}).$$

The *getExpTerm* function traverses all the OclExpression instances that constitute a constraint definition $\widetilde{c}$ by considering the containment relation that is defined by $<_c (\widetilde{c}, \widetilde{\text{OCL}})$, following a top-down strategy. The *getExpTerm* function is defined for the following object types of the OCL metamodel: IfExp, IterateExp, IteratorExp, LetExp, LiteralExp, OperationCallExp, PropertyCallExp,VariableExp, and TypeExp; as described in subsequent paragraphs.

IfExp

An IfExp instance defines an OCL expression that results in one of two alternative expressions, *thenExpression* and *elseExpression*, depending on the evaluated value of a *condition*. Note that both the thenExpression and the elseExpression are mandatory. The reason behind this is that an if expression should always result in a value, which cannot be guaranteed if the else part is left out. An IfExp is represented by means of the `if_then_else_fi` operator that is defined in the `BOOL` theory, by means of the mapping

$$
\begin{array}{l}
getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
\quad \texttt{if}\ \boxed{getExpTerm(\widetilde{e}.condition(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{then} \\
\quad\ \boxed{getExpTerm(\widetilde{e}.thenExpression(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\
\quad \texttt{else} \\
\quad\ \boxed{getExpTerm(\widetilde{e}.elseExpression(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\
\quad \texttt{fi} \\
\text{where } \widetilde{e} : \textsc{IfExp}
\end{array}
$$

IterateExp

An IterateExp instance represents an expression which evaluates its *body* expression for each element of a *source* collection. The result can be of any type and is defined by the *result* property. An IterateExp instance is represented in the $reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory by means of the mapping:

$$
\begin{array}{l}
getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
\quad \boxed{getVariableValueProjector(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}})}\texttt{(} \\
\quad\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> iterate (} \\
\quad\quad\quad \boxed{\widetilde{e}.result(\widetilde{c}).name}\ \texttt{=} \\
\quad\quad\quad\quad \boxed{getExpTerm(\widetilde{e}.result(\widetilde{c}).initExpression(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{|} \\
\quad\quad\quad \boxed{getBodyExpName(\widetilde{e}.body(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}})}\ \texttt{;} \\
\quad\quad\quad \boxed{getEnvironment(EnvVars,\ \widetilde{\mathcal{M}})}\ \texttt{;} \\
\quad\quad\quad \boxed{getPreConfigurationValue(\widetilde{e}.iterator(\widetilde{\mathcal{M}}),\ \widetilde{\mathcal{M}})} \\
\quad\quad \texttt{)} \\
\quad \texttt{)} \\
\text{where } \widetilde{e} : \textsc{IterateExp}
\end{array}
$$

where the function

$$getVariableValueProjector : [\![\textsc{IterateExp}]\!]_{\textsc{OclExp}\times\textsc{OCL}\times\textsc{MOF}} \longrightarrow OpNames$$

obtains the operator symbol that projects the value of the *result* variable, depending on its type, as follows:

$$\begin{aligned}
&getVariableValueProjector(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{getObjectVariableValue}\\
&\quad \text{when } \widetilde{e}.result(\widetilde{c}).type(\widetilde{\mathcal{M}}) : \textsc{Class}\\
&getVariableValueProjector(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{getEnumVariableValue}\\
&\quad \text{when } \widetilde{e}.result(\widetilde{c}).type(\widetilde{\mathcal{M}}) : \textsc{Enumeration}\\
&getVariableValueProjector(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{getBoolVariableValue}\\
&\quad \text{when } \widetilde{e}.result(\widetilde{c}).type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Boolean}}, \widetilde{\textsc{MOF}})\\
&getVariableValueProjector(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{getStringVariableValue}\\
&\quad \text{when } \widetilde{e}.result(\widetilde{c}).type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{String}}, \widetilde{\textsc{MOF}})\\
&getVariableValueProjector(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{getIntVariableValue}\\
&\quad \text{when } \widetilde{e}.result(\widetilde{c}).type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Integer}}, \widetilde{\textsc{MOF}})\\
&getVariableValueProjector(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}}) = \texttt{getFloatVariableValue}\\
&\quad \text{when } \widetilde{e}.result(\widetilde{c}).type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Real}}, \widetilde{\textsc{MOF}})
\end{aligned}$$

The function $getPreConfigurationValue : TypedVariables \longrightarrow Terms$ obtains a term or a variable that is used as the last argument of the operators `_->iterate(_|_;_)`, `_->_(_;_)` and `_->sortedBy(_;_;_)`. This function is defined as follows:

$$\begin{aligned}
&getPreConfigurationValue(\widetilde{e},\ \widetilde{\mathcal{M}}) = \texttt{model:ModelType}\{\mathcal{M}\}\\
&\quad \text{when } \widetilde{e}.type(\widetilde{\mathcal{M}}) : \textsc{Class}\\
&getPreConfigurationValue(\widetilde{e},\ \widetilde{\mathcal{M}}) = \texttt{nonePreConf\#Enum}\\
&\quad \text{when } \widetilde{e}.type(\widetilde{\mathcal{M}}) : \textsc{Enumeration}\\
&getPreConfigurationValue(\widetilde{e},\ \widetilde{\mathcal{M}}) = \texttt{nonePreConf\#Bool}\\
&\quad \text{when } \widetilde{e}.type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Boolean}}, \widetilde{\textsc{MOF}})\\
&getPreConfigurationValue(\widetilde{e},\ \widetilde{\mathcal{M}}) = \texttt{nonePreConf\#String}\\
&\quad \text{when } \widetilde{e}.type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{String}}, \widetilde{\textsc{MOF}})\\
&getPreConfigurationValue(\widetilde{e},\ \widetilde{\mathcal{M}}) = \texttt{nonePreConf\#Int}\\
&\quad \text{when } \widetilde{e}.type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Integer}}, \widetilde{\textsc{MOF}})\\
&getPreConfigurationValue(\widetilde{e},\ \widetilde{\mathcal{M}}) = \texttt{nonePreConf\#Float}\\
&\quad \text{when } \widetilde{e}.type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Real}}, \widetilde{\textsc{MOF}})
\end{aligned}$$

For example, the OCL expression that sums the integer values in a collection of integers can be, in OCL textual concrete syntax,

```
Set{1, 2, 3} -> iterate( i:  Integer, sum:  Integer = 0 | sum + i ).
```

The `IterateExp` that represents this expression is represented in the $reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory as the term

```
getIntVariableValue(Set{1, 2, 3} -> iterate (
    sum = 0 |
    integerSum ; empty-env ; nonePrConf#Int )
)
```

IteratorExp

An IteratorExp instance represents an expression that evaluates its *body* expression for each element of a *source* collection. It acts as a loop construct that iterates over the elements of its source collection and results in a value. We consider the *sortedBy* iterator operator as a special case, as discussed in Section 7.2. An IteratorExp instance is represented as a term in the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory as follows:

$$
\begin{array}{l}
\mathit{getExpTerm}(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{EnvVars}) = \\
\quad \boxed{\mathit{getExpTerm}(\widetilde{e}.\mathit{source}(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{EnvVars})}\ \texttt{->}\ \boxed{\widetilde{e}.\mathit{name}}\ \texttt{(} \\
\quad\quad \boxed{\mathit{getBodyExpName}(\widetilde{e}.\mathit{body}(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}})}\ \texttt{;} \\
\quad\quad \boxed{\mathit{getEnvironment}(\mathit{EnvVars},\ \widetilde{\mathcal{M}})}\ \texttt{;} \\
\quad\quad \boxed{\mathit{getPreConfigurationValue}(\widetilde{e}.\mathit{iterator}(\widetilde{\mathcal{M}}),\ \widetilde{\mathcal{M}})} \\
\quad \texttt{)} \\
\quad \text{when } \widetilde{e} : \textsc{IteratorExp}\ \wedge\ \widetilde{e}.\mathit{name} \neq \text{"}\mathit{sortedBy}\text{"}
\end{array}
$$

$$
\begin{array}{l}
\mathit{getExpTerm}(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{EnvVars}) = \\
\quad \boxed{\mathit{getExpTerm}(\widetilde{e}.\mathit{source}(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{EnvVars})}\ \texttt{->}\ \boxed{\widetilde{e}.\mathit{name}}\ \texttt{(} \\
\quad\quad \boxed{\mathit{getBodyExpName}(\widetilde{e}.\mathit{body}(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}})}\ \texttt{;} \\
\quad\quad \boxed{\mathit{getEnvironment}(\mathit{EnvVars},\ \widetilde{\mathcal{M}})}\ \texttt{;} \\
\quad\quad \boxed{\mathit{getPreConfigurationValue}(\widetilde{e}.\mathit{iterator}(\widetilde{\mathcal{M}}),\ \widetilde{\mathcal{M}})} \\
\quad \texttt{)} \\
\quad \text{when } \widetilde{e} : \textsc{IteratorExp}\ \wedge\ \widetilde{e}.\mathit{name} = \text{"}\mathit{sortedBy}\text{"}
\end{array}
$$

The *forAll* expression

```
A.allInstances() -> forAll(objA : A | objA.a.mod(2) <> 0)
```

is then defined by means of the term

```
... -> forAll( isOdd ; empty-env ; model:ModelType{Example}).
```

LetExp

A LetExp instance $\widetilde{e}$ is a special expression that defines a new variable with an initial value. A variable defined by a LetExp instance cannot change its value, which represents the evaluated value of an *initial expression.* The variable is visible in the *in* expression. A LetExp instance is not represented by any term. When the variable, $\widetilde{e}.\mathit{variable}(\widetilde{\mathcal{M}})$ that is declared in a *let* statement is used in an OCL expression, the value of the variable is obtained by generating the term that represents the initialization expression of the variable, i.e., $\widetilde{e}.\mathit{variable}(\widetilde{\mathcal{M}}).\mathit{initExpression}(\widetilde{\mathcal{M}})$, as shown below for the VariableExp object type.

LiteralExp

A LiteralExp instance represents an expression with no arguments producing a value. In general the result value is identical with the expression symbol. For example, a LiteralExp instance may represent the integer 1 or a literal string like ‘this is a LiteralExp’. When the literal value is not a collection value nor an undefined value it is represented as indicated in the following equations:

$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \boxed{\widetilde{e}.booleanSymbol}$$
$$\text{when } \widetilde{e} : \textsc{BooleanLiteralExp}$$
$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \boxed{\widetilde{e}.stringSymbol}$$
$$\text{when } \widetilde{e} : \textsc{StringLiteralExp}$$
$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \boxed{\widetilde{e}.integerSymbol}$$
$$\text{when } \widetilde{e} : \textsc{IntegerLiteralExp}$$
$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \boxed{\widetilde{e}.realSymbol}$$
$$\text{when } \widetilde{e} : \textsc{RealLiteralExp}$$
$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \boxed{\widetilde{e}.literalExp(\widetilde{c}).name}$$
$$\text{when } \widetilde{e} : \textsc{EnumLiteralExp}$$

When the literal value is an undefined value, the corresponding null constant is generated as follows:

$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \texttt{nullBool}$$
$$\text{when } \widetilde{e} : \textsc{NullLiteralExp} \ \wedge\ \widetilde{e}.type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Boolean}}, \widetilde{\mathrm{MOF}})$$
$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \texttt{nullString}$$
$$\text{when } \widetilde{e} : \textsc{NullLiteralExp} \ \wedge\ \widetilde{e}.type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{String}}, \widetilde{\mathrm{MOF}})$$
$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \texttt{nullInt}$$
$$\text{when } \widetilde{e} : \textsc{NullLiteralExp} \ \wedge\ \widetilde{e}.type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Integer}}, \widetilde{\mathrm{MOF}})$$
$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \texttt{nullFloat}$$
$$\text{when } \widetilde{e} : \textsc{NullLiteralExp} \ \wedge\ \widetilde{e}.type(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Real}}, \widetilde{\mathrm{MOF}})$$

$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \texttt{nullEnum}$$
$$\text{when } \widetilde{e} : \textsc{NullLiteralExp} \ \wedge\ \widetilde{e}.type(\widetilde{\mathcal{M}}) : \textsc{Enumeration}$$
$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \texttt{nullObject\#}\mathcal{M}$$
$$\text{when } \widetilde{e} : \textsc{NullLiteralExp} \ \wedge\ \widetilde{e}.type(\widetilde{\mathcal{M}}) : \textsc{Class}$$

When the literal value represents a collection value, the term that represents the collection is generated as follows:

$$
\begin{array}{l}
\mathit{getExpTerm}(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{EnvVars}) = \boxed{\mathit{getCollectionOpName}(\widetilde{e})}\{ \\
\quad \boxed{\mathit{getCollectionItem}(\widetilde{e}.\mathit{parts}(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{false})} \\
\quad \} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge\ \widetilde{e}.\mathit{part}(\widetilde{c}) \neq \varnothing \\
\mathit{getExpTerm}(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{EnvVars}) = \boxed{\mathit{getEmptyCollectionOpName}(\widetilde{e})}\texttt{\#}\mathcal{M} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge\ \widetilde{e}.\mathit{part}(\widetilde{c}) = \varnothing\ \wedge \\
\qquad \widetilde{e}.\mathit{elementType}(\widetilde{\mathcal{M}}) : \textsc{Class} \\
\mathit{getExpTerm}(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{EnvVars}) = \boxed{\mathit{getEmptyCollectionOpName}(\widetilde{e})}\texttt{\#Enum} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge\ \widetilde{e}.\mathit{part}(\widetilde{c}) = \varnothing\ \wedge \\
\qquad \widetilde{e}.\mathit{elementType}(\widetilde{\mathcal{M}}) : \textsc{Enumeration} \\
\mathit{getExpTerm}(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{EnvVars}) = \boxed{\mathit{getEmptyCollectionOpName}(\widetilde{e})}\texttt{\#Bool} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge\ \widetilde{e}.\mathit{part}(\widetilde{c}) = \varnothing\ \wedge \\
\qquad \widetilde{e}.\mathit{elementType}(\widetilde{\mathcal{M}}) = \mathit{root}(\widetilde{\textsc{Boolean}}, \widetilde{\text{MOF}}) \\
\mathit{getExpTerm}(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{EnvVars}) = \boxed{\mathit{getEmptyCollectionOpName}(\widetilde{e})}\texttt{\#String} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge\ \widetilde{e}.\mathit{part}(\widetilde{c}) = \varnothing\ \wedge \\
\qquad \widetilde{e}.\mathit{elementType}(\widetilde{\mathcal{M}}) = \mathit{root}(\widetilde{\textsc{String}}, \widetilde{\text{MOF}}) \\
\mathit{getExpTerm}(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{EnvVars}) = \boxed{\mathit{getEmptyCollectionOpName}(\widetilde{e})}\texttt{\#Int} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge\ \widetilde{e}.\mathit{part}(\widetilde{c}) = \varnothing\ \wedge \\
\qquad \widetilde{e}.\mathit{elementType}(\widetilde{\mathcal{M}}) = \mathit{root}(\widetilde{\textsc{Integer}}, \widetilde{\text{MOF}}) \\
\mathit{getExpTerm}(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ \mathit{EnvVars}) = \boxed{\mathit{getEmptyCollectionOpName}(\widetilde{e})}\texttt{\#Float} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge\ \widetilde{e}.\mathit{part}(\widetilde{c}) = \varnothing\ \wedge \\
\qquad \widetilde{e}.\mathit{elementType}(\widetilde{\mathcal{M}}) = \mathit{root}(\widetilde{\textsc{Real}}, \widetilde{\text{MOF}})
\end{array}
$$

where the function

$$\mathit{getCollectionOpName} : \mathit{CollectionLiteralExp} \rightsquigarrow \mathit{OpName}$$

obtains the operator symbol to define a set, an ordered set, a bag or a sequence as follows:

$$
\begin{array}{l}
\mathit{getCollectionOpName}(\widetilde{e}) = \texttt{Set} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge\ \widetilde{e}.\mathit{kind} = \mathit{set} \\
\mathit{getCollectionOpName}(\widetilde{e}) = \texttt{OrderedSet} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge\ \widetilde{e}.\mathit{kind} = \mathit{orderedset} \\
\mathit{getCollectionOpName}(\widetilde{e}) = \texttt{Bag} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge\ \widetilde{e}.\mathit{kind} = \mathit{bag} \\
\mathit{getCollectionOpName}(\widetilde{e}) = \texttt{Sequence} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge\ \widetilde{e}.\mathit{kind} = \mathit{sequence}
\end{array}
$$

The function

$$\mathit{getEmptyCollectionOpName} : \mathit{CollectionLiteralExp} \rightsquigarrow \mathit{OpName}$$

generates the corresponding constant to define an empty collection, depending on the collection type, as follows:

$$\begin{array}{l}
getEmptyCollectionOpName(\widetilde{e}) = \texttt{empty-set} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge \ \widetilde{e}.kind = set \\
getEmptyCollectionOpName(\widetilde{e}) = \texttt{empty-orderedset} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge \ \widetilde{e}.kind = orderedset \\
getEmptyCollectionOpName(\widetilde{e}) = \texttt{empty-bag} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge \ \widetilde{e}.kind = bag \\
getEmptyCollectionOpName(\widetilde{e}) = \texttt{empty-sequence} \\
\quad \text{when } \widetilde{e} : \textsc{CollectionLiteralExp} \ \wedge \ \widetilde{e}.kind = sequence
\end{array}$$

The function

$$\begin{array}{l}
getCollectionItem : T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), Sequence\{\mathcal{M}\}} \times [\![\mathrm{OCL}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \times [\![\textsc{Boolean}]\!]_{\mathrm{MOF}} \\
\quad \rightsquigarrow Terms
\end{array}$$

obtains a term of sort `Magma{T}` when the collection that is being represented as a term is a set or a bag, and a term of sort `OrderedMagma{T}` when the collection that is being represented as a term is an ordered set or a sequence. This function is defined for tuples of the form $(seq, \widetilde{c}, \widetilde{\mathcal{M}}, ordered)$ where *seq* is the sequence of values that defines the items of the collection literal that is being processed, $\widetilde{c}$ represents the constraint definition in which the literal is defined, $\widetilde{\mathcal{M}}$ is the metamodel definition for which the constraint definition $\widetilde{c}$ is meaningful, and *ordered* indicates whether the collection literal is ordered, i.e., it is of type ordered set or sequence, or not, i.e., it is of type set or bag. The *getCollectionItem* function is defined as follows:

$$\begin{array}{l}
getCollectionItem(seq,\ \widetilde{c},\ \widetilde{\mathcal{M}},\ ordered) = \\
\quad \boxed{getExpTerm(\widetilde{e}.item(\widetilde{c}), \widetilde{c},\ \widetilde{\mathcal{M}}, empty\text{-}env)}\ , \\
\quad \boxed{getCollectionItem(seq - Sequence\{\widetilde{e}\}, \widetilde{c},\ \widetilde{\mathcal{M}}, empty\text{-}env)} \\
\ \text{when } seq \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), Sequence\{\mathcal{M}\}} \ \wedge \ (seq \texttt{ -> } size) > 1 \ \wedge \ ordered = false \ \wedge \\
\ \widetilde{e} : \textsc{CollectionItem} \\
getCollectionItem(seq,\ \widetilde{c},\ \widetilde{\mathcal{M}},\ ordered) = \\
\quad \boxed{getExpTerm(\widetilde{e}.item(\widetilde{c}), \widetilde{c},\ \widetilde{\mathcal{M}}, empty\text{-}env)}\ :: \\
\quad \boxed{getCollectionItem(seq - Sequence\{\widetilde{e}\}, \widetilde{c},\ \widetilde{\mathcal{M}}, empty\text{-}env)} \\
\ \text{when } seq \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), Sequence\{\mathcal{M}\}} \ \wedge \ (seq \texttt{ -> } size) > 1 \ \wedge \ ordered = true \ \wedge \\
\ \widetilde{e} : \textsc{CollectionItem} \\
getCollectionItem(seq,\ \widetilde{c},\ \widetilde{\mathcal{M}},\ ordered) = \\
\quad \boxed{getExpTerm(\widetilde{e}.item(\widetilde{c}), \widetilde{c},\ \widetilde{\mathcal{M}}, empty\text{-}env)} \\
\ \text{when } seq \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}), Sequence\{\mathcal{M}\}} \ \wedge \ (seq \texttt{ -> } size) = 1 \ \wedge \\
\ \widetilde{e} : \textsc{CollectionItem}
\end{array}$$

OperationCallExp

An OperationCallExp instance refers to an operation, which is predefined for either the OCL collection types or an object type in $\widetilde{\mathcal{M}}$. The OCL predefined operators are defined as Operation instances in the metamodel definition $\widetilde{\textsc{OclStdLib}}$, such that $\textsc{OclStdLib}$: MOF. This metamodel is defined as a constant, because it is only used to be queried. Therefore, we do not need to pass it as an argument to the *getExpTerm* function. We do not consider operations that are defined for object types in the metamodel definition $\widetilde{\mathcal{M}}$. A OperationCallExp instance may contain a list of *argument* expressions if the operation is defined to have parameters. In this case, the number and types of the arguments must match the parameters. An OperationCallExp instance is represented as a term in which a predefined operator, which has been defined in the `OCL-COLLECTIONS{T :: TRIV}` theory, is used. The definitions of the *getExpTerm* function for each of the predefined operators of the OCL language are as follows:

- `count` :

$$\begin{array}{l}
getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> count(} \\
\qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\
\quad \texttt{)} \\
\text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset \wedge \\
\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}count\text{"}
\end{array}$$

- `excludes` :

$$\begin{array}{l}
getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> excludes(} \\
\qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\
\quad \texttt{)} \\
\text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset \wedge \\
\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}excludes\text{"}
\end{array}$$

- `excludesAll` :

$$\begin{array}{l}
getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> excludesAll(} \\
\qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\
\quad \texttt{)} \\
\text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset \wedge \\
\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}excludesAll\text{"}
\end{array}$$

- `includes` :

$$\begin{array}{l}
getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> includes(} \\
\qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\
\quad \texttt{)} \\
\text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset \wedge \\
\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}includes\text{"}
\end{array}$$

- `includesAll` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> includesAll(} \\ \qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\ \quad \texttt{)} \\ \text{when } \widetilde{e} : \text{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation \neq \emptyset \ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\text{OclStdLib}}).name = \text{"}includesAll\text{"} \end{array}$$

- `isEmpty` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> isEmpty} \\ \text{when } \widetilde{e} : \text{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation \neq \emptyset \ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\text{OclStdLib}}).name = \text{"}isEmpty\text{"} \end{array}$$

- `notEmpty` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> notEmpty} \\ \text{when } \widetilde{e} : \text{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation \neq \emptyset \ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\text{OclStdLib}}).name = \text{"}notEmpty\text{"} \end{array}$$

- `size` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> size} \\ \text{when } \widetilde{e} : \text{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation \neq \emptyset \ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\text{OclStdLib}}).name = \text{"}size\text{"} \end{array}$$

- `sum` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> sum} \\ \text{when } \widetilde{e} : \text{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation \neq \emptyset \ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\text{OclStdLib}}).name = \text{"}sum\text{"} \end{array}$$

- `-` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{--} \\ \qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\ \text{when } \widetilde{e} : \text{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation \neq \emptyset \ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\text{OclStdLib}}).name = \text{"} - \text{"} \end{array}$$

- `append` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> append(} \\ \qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\ \quad \texttt{)} \\ \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset\ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}append\text{"} \end{array}$$

- `asBag` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> asBag} \\ \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset\ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}asBag\text{"} \end{array}$$

- `asOrderedSet` : The `asOrderedSet` operator is not considered to avoid non-determinism, as discussed below.
- `asSequence` : The `asSequence` operator is not considered to avoid non-determinism, as discussed below.
- `asSet` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> asSet} \\ \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset\ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}asSet\text{"} \end{array}$$

- `at` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> at(} \\ \qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\ \quad \texttt{)} \\ \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset\ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}at\text{"} \end{array}$$

- `excluding` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> excluding(} \\ \qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\ \quad \texttt{)} \\ \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset\ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}excluding\text{"} \end{array}$$

- `first` :

$$
\begin{aligned}
&getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
&\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> first} \\
&\quad \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset \wedge \\
&\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}\mathit{first}\text{"}
\end{aligned}
$$

- `flatten` :

$$
\begin{aligned}
&getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
&\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> flatten} \\
&\quad \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset \wedge \\
&\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}\mathit{flatten}\text{"}
\end{aligned}
$$

- `including` :

$$
\begin{aligned}
&getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
&\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> including(} \\
&\qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\
&\quad\ \texttt{)} \\
&\quad \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset \wedge \\
&\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}\mathit{including}\text{"}
\end{aligned}
$$

- `indexOf` :

$$
\begin{aligned}
&getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
&\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> indexOf(} \\
&\qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\
&\quad\ \texttt{)} \\
&\quad \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset \wedge \\
&\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}\mathit{indexOf}\text{"}
\end{aligned}
$$

- `insertAt` :

$$
\begin{aligned}
&getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
&\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> insertAt(} \\
&\qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c})\ \texttt{->}\ at(1),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ ; \\
&\qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c})\ \texttt{->}\ at(2),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\
&\quad\ \texttt{)} \\
&\quad \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset \wedge \\
&\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}\mathit{insertAt}\text{"}
\end{aligned}
$$

- `intersection` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> intersection(} \\ \qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\ \quad \texttt{)} \\ \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation \ \neq\ \emptyset\ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}intersection\text{"} \end{array}$$

- `last` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> last} \\ \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation \ \neq\ \emptyset\ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}last\text{"} \end{array}$$

- `prepend` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> prepend(} \\ \qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\ \quad \texttt{)} \\ \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation \ \neq\ \emptyset\ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}prepend\text{"} \end{array}$$

- `union` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> union(} \\ \qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\ \quad \texttt{)} \\ \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation \ \neq\ \emptyset\ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}union\text{"} \end{array}$$

The operators that are defined for the OCL special types, *OclAny* and *OclType*, can be used in an algebraic OCL expression as follows:

- `=` :

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{=} \\ \qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\ \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation \ \neq\ \emptyset\ \wedge \\ \qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{" = "} \end{array}$$

- `<>` :

$$
\begin{aligned}
&getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
&\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{<>} \\
&\qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\
&\quad \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset\ \wedge \\
&\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"} <> \text{"}
\end{aligned}
$$

- `oclIsUndefined` :

$$
\begin{aligned}
&getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
&\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> oclIsUndefined} \\
&\quad \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset\ \wedge \\
&\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}oclIsUndefined\text{"}
\end{aligned}
$$

- `oclIsKindOf` :

$$
\begin{aligned}
&getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
&\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> oclIsKindOf(} \\
&\qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\
&\quad \texttt{)} \\
&\quad \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset\ \wedge \\
&\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}oclIsKindOf\text{"}
\end{aligned}
$$

- `oclIsTypeOf` :

$$
\begin{aligned}
&getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
&\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> oclIsTypeOf(} \\
&\qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\
&\quad \texttt{)} \\
&\quad \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset\ \wedge \\
&\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}oclIsTypeOf\text{"}
\end{aligned}
$$

- `oclAsType` :

$$
\begin{aligned}
&getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\
&\quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \texttt{-> oclAsType(} \\
&\qquad \boxed{getExpTerm(\widetilde{e}.argument(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\
&\quad \texttt{)} \\
&\quad \text{when } \widetilde{e} : \textsc{OperationCallExp} \ \wedge\ \widetilde{e}.referredOperation\ \neq\ \emptyset\ \wedge \\
&\qquad \widetilde{e}.referredOperation(\widetilde{\textsc{OclStdLib}}).name = \text{"}oclAsType\text{"}
\end{aligned}
$$

PropertyCallExp

A PropertyCallExp instance is a reference to a Property instance that is defined in the metamodel definition $\widetilde{\mathcal{M}}$. The term that represents a PropertyCallExp instance in an OCL expression is defined by means of the *getExpTerm* function as follows:

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \ . \\ \quad \boxed{\widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).name} \\ \text{when } \widetilde{e} : \textsc{PropertyCallExp}\ \wedge\ \widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).type(\widetilde{\mathcal{M}}) : \textsc{DataType} \end{array}$$

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.source(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)}\ \ . \\ \quad \boxed{\widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).name}\texttt{(model:ModelType\{}\mathcal{M}\texttt{\})} \\ \text{when } \widetilde{e} : \textsc{PropertyCallExp}\ \wedge\ \widetilde{e}.referredProperty(\widetilde{\mathcal{M}}).type(\widetilde{\mathcal{M}}) : \textsc{Class} \end{array}$$

Given a model definition $\widetilde{M}$, such that $\widetilde{M}$: RDBMS, where RDBMS is the metamodel that is defined in Fig. 6.7, to query the value-typed property `name` of a Table instance `t`, such that `t` $\in \widetilde{M}$, we use the term `t . name`. To query the object-typed property `schema` of the object `t`, we can use either the term `t . schema`, obtaining an object identifier, or the term `t . schema(`$\widetilde{M}$`)`, obtaining a Schema instance, in case it is defined in $\widetilde{M}$.

VariableExp

A VariableExp instance represents an expression that consists of a reference to a variable. The variables that can be referenced are: the *self* variable that contains the *contextual instance*; the *iterator* variable of a loop operator; the *result* variable of an *iterate* operator; or a *Let* variable. When a VariableExp instance is projected as a term by means of the *getExpTerm* function, the referred variable has always been previously defined in the environment of the expression, so that the value of the variable has been already computed, except for one case: for *Let* variables. For a *Let* variable, the *getExpTerm* represents its value by processing the initial expression that is attached to it. *Let* variables are not added to the environment of the OCL expressions, so that a VariableExp instance, whose referenced variable is not contained in the environment, can only refer to a variable of this kind. The term that represents a VariableExp instance is a variable that is given by the following equations:

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getSortedVariableValue(\widetilde{e}.referredVariable(\widetilde{c}), \widetilde{\mathcal{M}})} \\ \text{when } \widetilde{e} : \textsc{VariableExp}\ \wedge\ \widetilde{e}.referredVariable(\widetilde{c}) \in EnvVars \end{array}$$

$$\begin{array}{l} getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \\ \quad \boxed{getExpTerm(\widetilde{e}.referredVariable(\widetilde{c}).initExpression(\widetilde{c}),\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars)} \\ \text{when } \widetilde{e} : \textsc{VariableExp}\ \wedge\ \widetilde{e}.referredVariable(\widetilde{c}) \notin EnvVars \end{array}$$

TypeExp

A TypeExp instance is an expression used to refer to an existing meta-type within an expression. It is used in particular to pass the reference of the meta-type when invoking the operations *allInstances*, *oclIsKindOf*, *oclIsTypeOf*, and *oclAsType*. The term that represents a TypeExp instance is a constant of type `Cid#`$\mathcal{M}$

when the type refers to an object type. The function *getExpTerm* is defined, for a TypeExp instance as follows:

$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \boxed{\widetilde{e}.referredType(\widetilde{\mathcal{M}}).name}$$
$$\text{when } \widetilde{e} : \textsc{TypeExp} \ \wedge\ \widetilde{e}.referredType(\widetilde{\mathcal{M}}) : \textsc{Class}$$

$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \texttt{Bool}$$
$$\text{when } \widetilde{e} : \textsc{TypeExp} \ \wedge\ \widetilde{e}.referredType(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Boolean}}, \widetilde{\text{MOF}})$$

$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \texttt{String}$$
$$\text{when } \widetilde{e} : \textsc{TypeExp} \ \wedge\ \widetilde{e}.referredType(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{String}}, \widetilde{\text{MOF}})$$

$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \texttt{Int}$$
$$\text{when } \widetilde{e} : \textsc{TypeExp} \ \wedge\ \widetilde{e}.referredType(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Integer}}, \widetilde{\text{MOF}})$$

$$getExpTerm(\widetilde{e},\ \widetilde{c},\ \widetilde{\mathcal{M}},\ EnvVars) = \texttt{Float}$$
$$\text{when } \widetilde{e} : \textsc{TypeExp} \ \wedge\ \widetilde{e}.referredType(\widetilde{\mathcal{M}}) = root(\widetilde{\textsc{Float}}, \widetilde{\text{MOF}})$$

Note that in our specification these operators do not provide support for enumerations.

7.3.4 Name Strategy

The names that correspond to Class instances, EnumerationLiteral instances, and Property instances are used to define operator symbols in a *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory as shown before. These names are structured by taking into account the containment relation that is defined for a metamodel definition $\widetilde{\mathcal{M}}$ as $<_c (\widetilde{\mathcal{M}}, \widetilde{\text{MOF}})$. This strategy to structure names is achieved by means of the function *buildName* that is defined at the end of Section 6.3. Therefore, we avoid name collisions when two properties have the same name within different object types, for example. We have omitted this detail in the definition of the *getExpTheory* and *getExpTerm* functions for the sake of a simpler exposition.

7.3.5 Complete Example

The constraint that is defined as example for the relational metamodel in Section 2.3, i.e.,

```
context ForeignKey:
inv:
    if (self.column->size() = self.refersTo.column->size()) then
        self.column->forAll(c:Column |
            self.refersTo.column-> at(self.column->indexOf(c)).type
            = c.type
        )
    else
        false
    endif
```

constitutes the single constraint that is defined in the set $\widetilde{\mathcal{C}}_{\text{RDBSMS}}$ of OCL constraints. The type $(\text{RDBMS}, \mathcal{C}_{\text{RDBSMS}})$ is defined in the *reflect*$(\widetilde{\text{RDBMS}}, \widetilde{\mathcal{C}}_{\text{RDBSMS}})$ theory by means of the following conditional membership:

```
cmb model:ModelType{rdbms} : ConsistentModelType{rdbms}
if
    (
        rdbms/ForeignKey . allInstances( model:ModelType{rdbms} )
            -> forAll ( inv1 ; empty-env ; model:ModelType{rdbms} )
    ) = true .
```

where the operators that permit navigating through the object-typed properties that are used in the invariant are:

```
op rdbms/Key/column : -> Fun{rdbms} .
eq < OID:Oid : CID:rdbms/Column | PS:PropertySet#rdbms,
    rdbms/Key/column : Value:Collection+{Oid} > . rdbms/Key/column
    = Value:Collection+{Oid} .
eq < OID:oid#rdbms/Column : CID:rdbms/Column | PS:PropertySet#rdbms,
    rdbms/Key/column : Value:Collection+{Oid} >
        . rdbms/Key/column(model:ModelType{rdbms})
    = search(Value:Collection+{Oid}, model:ModelType{rdbms}) .

op rdbms/ForeignKey/column : -> Fun{rdbms} .
eq < OID:Oid : CID:rdbms/Column | PS:PropertySet#rdbms,
    rdbms/ForeignKey/column : Value:Collection+{Oid} >
    . rdbms/ForeignKey/column
    = Value:Collection+{Oid} .
eq < OID:oid#rdbms/Column : CID:rdbms/Column | PS:PropertySet#rdbms,
    rdbms/ForeignKey/column : Value:Collection+{Oid} >
        . rdbms/ForeignKey/column(model:ModelType{rdbms})
    = search(Value:Collection+{Oid}, model:ModelType{rdbms}) .

op rdbms/ForeignKey/refersTo : -> Fun{rdbms} .
eq < OID:Oid : CID:rdbms/Column | PS:PropertySet#rdbms,
    rdbms/ForeignKey/refersTo : Value:Collection+{Oid} >
    . rdbms/ForeignKey/refersTo
    = Value:Collection+{Oid} .
eq < OID:oid#rdbms/Column : CID:rdbms/Column | PS:PropertySet#rdbms,
    rdbms/ForeignKey/refersTo : Value:Collection+{Oid} >
        . rdbms/ForeignKey/refersTo(model:ModelType{rdbms})
    = search(Value:Collection+{Oid}, model:ModelType{rdbms}) .

op rdbms/Column/type : -> Fun{rdbms} .
eq < OID:Oid : CID:rdbms/Column | PS:PropertySet#rdbms,
    rdbms/Column/type : Value:Collection+{Oid} > . rdbms/Column/type
    = Value:Collection+{Oid} .
eq < OID:oid#rdbms/Column : CID:rdbms/Column | PS:PropertySet#rdbms,
    rdbms/Column/type : Value:Collection+{Oid} >
        . rdbms/Column/type(model:ModelType{rdbms})
    = search(Value:Collection+{Oid}, model:ModelType{rdbms}) .
```

and the OCL expressions are defined by means of the *reflect* function as follows:

```
op self : -> VariableName [ctor] .
op inv1 : -> BoolBody{rdbms} [ctor] .
eq selfValue:Collection+{rdbms} . inv1 (
    empty-env ; model:ModelType{rdbms}
) = (
    if (
        ((selfValue:Collection+{rdbms}
            . rdbms/ForeignKey/column ( model:ModelType{rdbms} )
        ) -> size)
        =
        (
            (selfValue:Collection+{rdbms}
                . rdbms/ForeignKey/refersTo ( model:ModelType{rdbms} )
                . rdbms/Key/column ( model:ModelType{rdbms} )
            ) -> size
        )
    )
    then
        ((selfValue:Collection+{rdbms}
            . rdbms/ForeignKey/column ( model:ModelType{rdbms} ))
            -> forAll (
                inv::body0 ;
                ? self = selfValue:Collection+{rdbms} ;
                rdbmsModel
            )
        )
    else
        false
    fi
) .
eq self::0 . inv1 ( E:Environment ; model:ModelType{rdbms} )
    = false [owise] .

op inv::body0 : -> BoolBody{rdbms} [ctor].
eq cValue:Collection+{rdbms} . inv::body0 (
    ? self = selfValue:Collection+{rdbms} ; model:ModelType{rdbms}
) =
(
    (
        (
            (selfValue:Collection+{rdbms}
                . rdbms/ForeignKey/refersTo ( model:ModelType{rdbms} )
                . rdbms/Key/column ( model:ModelType{rdbms} )
            )
            -> at(
                (
                    (selfValue:Collection+{rdbms}
                    . rdbms/ForeignKey/column ( model:ModelType{rdbms} ))
                    -> indexOf( cValue:Collection+{rdbms} )
                )
            )
        ) . rdbms/Column/type
    )
    =
    (cValue:Collection+{rdbms} . rdbms/Column/type)
) .
eq cValue:Collection+{rdbms}
    . inv::body0 ( E:Enviroment ; model:ModelType{rdbms} ) = false [owise].
```

7.4 Algebraic Semantics of the Constrained Conformance Relation

Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the model type $[\![\mathcal{M}]\!]_{\text{MOF}}$ is defined in the $\mathit{reflect}_{\text{MOF}}(\widetilde{\mathcal{M}})$ theory. The model type $[\![\mathcal{M}]\!]_{\text{MOF}}$ is preserved in the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory by means of the subtheory inclusion

$$\mathit{reflect}_{\text{MOF}}(\widetilde{\mathcal{M}}) \subseteq \mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}),$$

where the constrained model type $[\![(\mathcal{M}, \mathcal{C})]\!]_{\text{MOF}}$ is defined as a subset of the model type $[\![\mathcal{M}]\!]_{\text{MOF}}$, i.e., $[\![(\mathcal{M}, \mathcal{C})]\!]_{\text{MOF}} \subseteq [\![\mathcal{M}]\!]_{\text{MOF}}$. Assuming that the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory is confluent, terminating and pre-regular, the semantics of the constrained model type $[\![(\mathcal{M}, \mathcal{C})]\!]_{\text{MOF}}$ is defined by using the initial algebra of the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory as

$$\boxed{[\![(\mathcal{M}, \mathcal{C})]\!]_{\text{MOF}} = T_{\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}), \mathit{ConsistentModelType}\{\mathcal{M}\}};}$$

the OCL constraint satisfaction is defined as

$$\boxed{\widetilde{M} \models \mathcal{C} \iff \widetilde{M} \in [\![(\mathcal{M}, \mathcal{C})]\!]_{\text{MOF}};}$$

and the constrained conformance relation $\widetilde{M} : (\mathcal{M}, \mathcal{C})$ is defined as

$$\boxed{\widetilde{M} : (\mathcal{M}, \mathcal{C}) \Longleftrightarrow \widetilde{M} \in [\![(\mathcal{M}, \mathcal{C})]\!]_{\mathrm{MOF}}.}$$

In addition, the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory constitutes a formal realization as a theory in MEL of the metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. The theory $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ satisfies some executability requirements: Church-Rosser, termination and pre-regularity. Therefore, the metamodel specification realization $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ is executable, providing a formal decision procedure for the OCL constraint satisfaction relation, and is a MEL theory with initial algebra semantics, providing the algebraic semantics for the types that are defined as data in $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$.

Given the data type *Module* whose terms, of the form $\overline{(S, <, \Omega, E \cup A)}$, metarepresent MEL theories of the form $(S, <, \Omega, E \cup A)$, and the data type *Term*, whose terms, of the form $\overline{t}$, metarepresent terms t for a given MEL theory $(S, <, \Omega, E \cup A)$, the representation of the *getExpTheory* and *getExpTerm* functions can be easily embedded into MEL by using its reflective features as the following equationally-defined functions:

$$\begin{aligned}
&\overline{\mathit{getExpTheory}} :\\
&\qquad \mathit{Object\#OCL} \times \mathit{ModelType}\{\mathit{OCL}\} \times \mathit{ModelType}\{\mathit{MOF}\} \times \mathit{Configuration}\{\mathit{OCL}\}\\
&\qquad \rightsquigarrow \mathit{Module}; \text{ and}\\
&\overline{\mathit{getTermTheory}} :\\
&\qquad \mathit{Object\#OCL} \times \mathit{ModelType}\{\mathit{OCL}\} \times \mathit{ModelType}\{\mathit{MOF}\} \times \mathit{Configuration}\{\mathit{OCL}\}\\
&\qquad \rightsquigarrow \mathit{Term}.
\end{aligned}$$

In our formal framework, only part of the *reflect* function has been specified: the $\mathit{reflect}_{MOF}$ function, which has been defined in Section 6. The formal definition of the *reflect* function, based on the *getExpTheory* and *getExpTerm* functions, which has been provided in this Section, is based on the experience with a previous prototype that provides OCL constraint validation by using Maude as a functional programming language [11]. The mathematical definition of the *reflect* function permits using the MOF reflection mechanism for theoretical purposes. The complete specification of the *reflect* function is still not available in our formal MOF framework, although large portions of its specification have already been implemented in Maude. Finishing the rest of the *reflect* implementation is straightforward, since the formal semantics of the *reflect* function has already been provided in this work.

In a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, $\widetilde{\mathcal{M}}$ is a metamodel that is defined by using the EMOF metamodel and $\widetilde{\mathcal{C}}$ is a finite set of OCL constraints that are defined by using the OCL metamodel. Therefore, any metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ that is defined by means of the standard metamodels EMOF and OCL can be formally defined by means of the *reflect* function in our framework. Both the EMOF and the OCL standard specifications provide a finite set of OCL constraints for the metamodel definitions $\widetilde{\mathrm{MOF}}$ and $\widetilde{\mathrm{OCL}}$, resulting in the metamodel specification definitions $(\mathrm{MOF}, \mathcal{C}_{\mathrm{MOF}})$ and $(\mathrm{OCL}, \mathcal{C}_{\mathrm{OCL}})$, respectively. A metamodel definition $\widetilde{\mathcal{M}}$ satisfies the OCL constraints that are defined for the MOF metamodel iff $\widetilde{\mathcal{M}} : (\mathrm{MOF}, \mathcal{C}_{\mathrm{MOF}})$, and, likewise, an OCL constraint definition $\widetilde{c}$ satisfies the constraints that are defined for the OCL metamodel definition iff $\widetilde{c} : (\mathrm{OCL}, \mathcal{C}_{\mathrm{OCL}})$. Note that the

$$\mathit{reflect} : \mathit{SpecMOF} \rightsquigarrow \mathit{SpecMEL}$$

is totally defined for pairs $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ such that

$$\begin{aligned}
&\widetilde{\mathcal{M}} : (\mathrm{MOF}, \mathcal{C}_{\mathrm{MOF}}) \wedge\\
&\widetilde{\mathcal{C}} \in \mathcal{P}_{\mathit{fin}}([\![(\mathrm{OCL}, \mathcal{C}_{\mathrm{OCL}})]\!]_{\mathrm{MOF}}) \wedge\\
&\forall \widetilde{c}(\widetilde{c} : (\mathrm{OCL}, \mathcal{C}_{\mathrm{OCL}}) \;\wedge\; \widetilde{c} \in \mathcal{C}_{\mathrm{OCL}} \;\wedge\; \mathit{root}(\widetilde{c}, \widetilde{\mathrm{OCL}}) : \textsc{OclConstraint} \;\rightarrow\\
&\qquad \mathit{root}(\widetilde{c}, \widetilde{\mathrm{OCL}}).\mathit{constrainedElement}(\widetilde{\mathcal{M}}) \neq \varnothing\\
&).
\end{aligned}$$

7.4.1 Discussion: Non-Determinism in OCL Expressions

Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the semantics of the metamodel specification $(\mathcal{M}, \mathcal{C})$ is provided by the carrier of the sort $\mathit{ConsistentModelType}\{\mathcal{M}\}$ in the initial algebra of the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$

theory. The constrained model type $[\![(\mathcal{M}, \mathcal{C})]\!]_{\text{MOF}}$ is provided by means of the conditional membership that is defined for the OCL constraint satisfaction relation. However, in order to ensure that all chains of equational simplification in the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory end in a unique canonical form (modulo associativity and commutativity), the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory must satisfy some executability requirements: the theory must be Church-Rosser, terminating and pre-regular.

Non-determinism can be introduced in OCL expressions in several ways: by means of OCL operators as indicated in [94], and by means of user-defined expressions. On the one hand, several predefined operators of the OCL language are non-deterministic: `any`, `asOrderedSet` and `asSequence`. The `asOrderedSet` and `asSequence` operators transform a non-ordered collection into an ordered set and to a sequence, respectively. For example, the expression `Set{1,2}->asOrderedSet` may result in either the collection literal `OrderedSet{1 :: 2}` or the collection literal `OrderedSet{2 :: 1}`. In addition, the operator `any` can be seen as an abbreviation for `asSequence` concatenated with `first`, another library operator which yields the first element of a sequence if the sequence has at least one element and an undefined value otherwise. Since the operator `asSequence` is non-deterministic, the operator `any` is not either.

This source of non-determinism cannot be specified by means of a function in a natural way. If these functions were specified in MEL, the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory would not be confluent. Therefore, an invariant, which contains non-deterministic constructs, that is evaluated over a model definition might yield more than one result, for example, *true* and *false*. An invariant like this is completely useless. Without loss of expressivity, we have omitted these operators from the OCL specification. When a non-ordered collection needs to be ordered, we use the *sortedBy* operator. For example, the expression `Set{1,2} -> sortedBy( i : Integer | i)` always result in the collection literal `OrderedSet{1 :: 2}`. On the other hand, the evaluation of OCL expressions may also lead to non-deterministic results as stated in [93]. For example, we can concatenate the strings that belong to a set by means of the following OCL expression:

```
Set{'a','b','c'} -> iterate(
    s : String;
    result : String = '' | result.concat(s) )
```

The resulting value of this expression can be any string that is constituted by a permutation of the characters 'a', 'b' and 'c'. Non-determinism can be avoided in this case, by ordering non-sorted collections before applying an OCL expression that may lead to non-deterministic results. In our formal framework only non-deterministic OCL expressions can be used to define the constraints in a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. Otherwise, the *reflect*$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory does not satisfy the Church-Rosser requirement. The *user* must take into account that *non-deterministic OCL expressions are not allowed.*

7.4.2 Unspecified Part of the OCL Language

In our approach, OCL is used to define the static semantics of a specific language using a metamodeling approach, for either a domain specific language or a general purpose language. Therefore, we have omitted the object types of the OCL metamodel that permit specifying constraints over the dynamics of a specific UML model. In our approach, only OCL expressions that are defined as invariants over object type definitions, in a specific metamodel definition, are taken into account. Other concepts of the OCL language that we have not considered yet are:

- *Only predefined OCL operators* can be used in OCL expressions. User-defined object type operations are not taken into account.
- *Tuple types* are not supported.
- *Association class navigation* is not supported, because the EMOF metamodel does not provide support for association classes.
- *Collection literals* cannot be defined by using number ranges.
- *Loop iterators* can only use a single iterator variable. In the OCL language the expression

  ```
  Set{1,2,3} -> forAll(i1,i2|
      ((Set{1,2,3} -> excluding(i1)) -> includes(i2)) implies i1<>i2
  ),
  ```

  which has two iterator variables, `i1` and `i2`, can be equivalently defined as

```
Set{1,2,3} -> forAll(i1|
    Set{1,2,3} -> forAll( i2 |
        ((Set{1,2,3} -> excluding(i1)) -> includes(i2))
            implies i1<>i2
    )
).
```

- The *oclIsTypeOf*, *oclIsKindOf*, and *oclAsType* operations are not specified for enumerations.

Chapter 8

Formalizing the MOF Reflection Facilities

Broadly speaking, reflection is the capacity to represent entities that have a formal semantics at a base level, such as types, as data at a metalevel. Reflection is a very powerful computational feature because metalevel entities, once metarepresented, can be computationally manipulated and transformed. Reflection was defined in a general way by Brian Smith [95]:

"An entity's integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on and deals with its primary subject matter."

In programming languages, the incarnation of this definition appears as follows [96]:

"Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection *and* intercession. *Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability of a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called* reification.*"*

Reflection is then usually achieved in three steps: *reification*, a program at the base level is represented as metadata that can be queried and, probably, manipulated at the metalevel; *absorption*, the tasks that change the program at the base level are performed by means of the operations of the metalevel, manipulating the metarepresentation of the program; *reflection*, once the metarepresentation of a program is changed, the new program is obtained at the base level again. There are several kinds of reflection [97]:

- *Introspection:* When a program has the capability to look at itself as data, and thereby to reason about it. This is usually achieved by means of a reification process. In our approach, introspection is achieved by the fact that the operators of a metamodel realization $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ can use the metamodel definition $\widetilde{\mathcal{M}}$.
- *Structural Reflection:* When a program has, in addition to introspection, the capability of modify its static semantics. In our approach, this is achieved by modifying a metamodel definition $\widetilde{\mathcal{M}}$, which metarepresents a metamodel realization $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ at the metalevel, and then reflecting it back again to the base level.
- *Behavioral Reflection:* When a program has, in addition to introspection and structural reflection, the capability of modifying its dynamic semantics. Since the MOF meta-metamodel does not permit defining the dynamic semantics of metamodels, we do not consider this kind of reflection.

In our approach, a metamodel definition $\widetilde{\mathcal{M}}$ constitutes the metarepresentation of a metamodel realization $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$, so that $\mathit{reify}(\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})) = \widetilde{\mathcal{M}}$, where the $\mathit{reflect}_{\mathrm{MOF}}$ and reify functions constitute the metarepresentation functions. In this section, we focus on the support for metarepresented entities manipulation that is provided in the MOF standard, which can be applied to both metamodel definitions $\widetilde{\mathcal{M}}$ and model definitions $\widetilde{M}$.

In the metamodel definition $\widetilde{\text{MOF}}$, the MOF reflection facilities are mainly provided by the OBJECT object type. The OBJECT object type is defined as supertype of any object type that is defined in a metamodel definition $\widetilde{\mathcal{M}}$, including $\widetilde{\text{MOF}}$ itself. Therefore, any object that can be defined in a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, is an instance of the OBJECT object type. Note that a metamodel definition $\widetilde{\mathcal{M}}$ is also a model definition $\widetilde{M}$, where $\widetilde{\mathcal{M}}$: MOF. The object type OBJECT provides a set of operations that permits the manipulation of the properties of a specific OBJECT instance.

In this section, we give an algebraic semantics for the OBJECT object type, and its operations, providing full support for reflection in the MOF framework. At the M2 level of the MOF framework, the reification mechanism, which is not provided in the MOF standard, is embodied by the *reify* function in our approach. The *reify* function permits metarepresenting a metamodel realization $\mathit{reflect}_{\text{MOF}}(\widetilde{\mathcal{M}})$ of the base level as a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF, at the metalevel. $\widetilde{\mathcal{M}}$ is a collection of typed objects, where each object is an instance of an object type of the MOF meta-metamodel. The OBJECT object type introduces an untyped level where metamodel definitions $\widetilde{\mathcal{M}}$ can be represented as collections of untyped objects, i.e., as OBJECT instances. The *instanceOf* relation is also represented as data at the metalevel, so that the manipulation of $\widetilde{\mathcal{M}}$ can be done independently of the object types that are defined in the realization of the MOF meta-metamodel, which is the $\mathit{reflect}_{\text{MOF}}(\widetilde{\text{MOF}})$ theory in our approach. If we move one level down in the MOF framework to the M1 level, the untyped level is also present for model definitions $\widetilde{M}$.

In subsequent sections, we present a brief description of the OBJECT object type and its operations, and its representation in our algebraic MOF framework by extending the infrastructure of theories that has been developed in Sections 6 and 7. We define the algebraic semantics of the OBJECT object type and the semantics of metarepresented model types. The semantics of the operations of the OBJECT object type are defined taking into account the semantics of the OCL types, so that OBJECT operations can also be used in terms that represent OCL expressions.

8.1 Informal Introduction to the MOF Reflective Facilities

In the MOF framework, any object that belongs to a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, can be manipulated by means of the operations of the OBJECT object type. These operations can be applied without any *realization* of the metamodel definition $\widetilde{\mathcal{M}}$. The OBJECT object type is defined in the MOF::REFLECTION package, shown in Fig. 8.1, of the metamodel definition $\widetilde{\text{MOF}}$. To define collections of objects, the MOF standard provides the REFLECTIVECOLLECTION and REFLECTIVESEQUENCE object types in the MOF::COMMON package, shown in Fig. 8.2, of the metamodel definition $\widetilde{\text{MOF}}$. In the MOF standard, the ELEMENT object type is defined as a supertype of both the OBJECT and REFLECTCOLLECTION object types. An ELEMENT instance may represent either an object, a data type value (string, integer,...), or a collection of ELEMENT instances. However, the standard does not indicate how primitive types are related to the ELEMENT object type.

In this section, we provide a brief description of the OBJECT object type, and its operations, which constitute the reflective capabilities of a MOF framework. These operations are defined in an informal way in the MOF specification. We illustrate them by using examples in an object-oriented programming style.

The OBJECT object type is defined as supertype of any object type that may be defined in a metamodel definition $\widetilde{\mathcal{M}}$, including $\widetilde{\text{MOF}}$ itself. Therefore, any object type inherits its operations, which are informally defined in the MOF standard, in an object-oriented programming style, as follows:

`getMetaClass(): Class`. Returns the class that describes this object. For example, the TABLE object type that is defined as a tree of objects, $\widetilde{\text{TABLE}}$, such that $\widetilde{\text{TABLE}}$: MOF, in the metamodel definition $\widetilde{\text{RDBMS}}$, contains a CLASS instance, which we denote by `TableClass`, such that `TableClass` = $\mathit{root}(\widetilde{\text{TABLE}}, \widetilde{\text{MOF}})$. Let `PersonTable` denote a TABLE instance, such that `PersonTable` : TABLE. The metaclass of the `PersonTable` object is obtained by means of the following expression: `PersonTable.getMetaClass()`, which results in the CLASS instance `TableClass`.

`container(): Object`. Returns the parent container of this element if any. Returns `null` if there is no containing element.

`equals(element: Element): Boolean`. Determines if the element argument is equal to a given Object instance. For instances of `Class`, returns true if the argument object and this Object instance are references to the same Object.

`set(property: Property, element: Element)`. If the property has multiplicity upper bound = 1, `set()` atomically updates the value of the property to the `element` parameter. If the property multiplicity

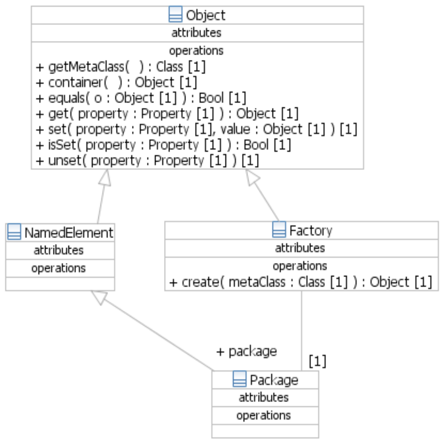

Object
attributes
operations
+ getMetaClass( ) : Class [1]
+ container( ) : Object [1]
+ equals(o : Object [1]) : Bool [1]
+ get(property : Property [1]) : Object [1]
+ set(property : Property [1], value : Object [1]) [1]
+ isSet(property : Property [1]) : Bool [1]
+ unset(property : Property [1]) [1]
NamedElement
attributes
operations
Factory
attributes
operations
+ create(metaClass : Class [1]) : Object [1]
+ package
[1]
Package
attributes
operations

Figure 8.1: The MOF Reflection API

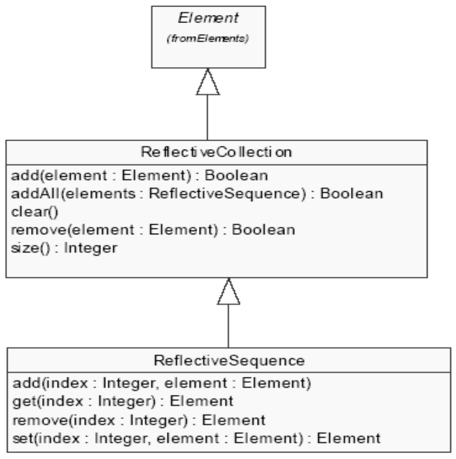

Element
(fromElements)
ReflectiveCollection
add(element : Element) : Boolean
addAll(elements : ReflectiveSequence) : Boolean
clear()
remove(element : Element) : Boolean
size() : Integer
ReflectiveSequence
add(index : Integer, element : Element)
get(index : Integer) : Element
remove(index : Integer) : Element
set(index : Integer, element : Element) : Element

Figure 8.2: The MOF Reflection API

upper bound is > 1, the `element` argument must be a collection of values. An exception is thrown in the following cases:

- The parameter property is not a member of the class that is obtained from `getMetaClass()`.
- The parameter `element` is not an instance of the type of the parameter property and the parameter property has multiplicity upper bound $= 1$.
- The parameter `element` is not a collection of values and the parameter property has multiplicity upper bound > 1.
- The parameter `element` is null, the parameter property is of type Class, and the multiplicity lower bound is $\geqslant 1$.

For the example, we can set the attribute "name" of the object `PersonTable` in an object-oriented way by invoking:

```
PersonTable.set(TableNameProperty, "Person")
```

where `TableNameProperty` corresponds to an `Object` instance in the metamodel whose `metaClass` is the MOF `Property` class, and it belongs to the collection of properties of the `TableClass` object. We can create new columns for the table by instantiating the `Column` class and by initializing the properties of the new instances:

```
Object nameColumn = create(ColumnClass)
nameColumn.set(ColumnNameProperty, "name")
nameColumn.set(ColumnTypeProperty, RDataType.VARCHAR)
Object ageColumn = create(ColumnClass)
ageColumn.set(ColumnNameProperty, "Age")
ageColumn.set(ColumnTypeProperty, RDataType.NUMBER)
```

where `create(class : Class)` is an operator of the FACTORY object type that creates a new instance, given the CLASS instance `class` that represents the root of an object type definition. To add the attributes that have been created above to the class `Person` in an object-oriented programming style, we obtain its collection of properties and we add the new attributes to it:

```
ReflectiveSequence rs = new ReflectiveSequence()
rs.add(nameColumn)
rs.add(ageColumn)
PersonTable.set(ColumnProperty, rs)
```

where `ColumnProperty` is an `Object` instance that represents the `column` property of the `Table` class in the relational metamodel.

`get(property: Property): Element.` Gets the value of the given property. If the property has multiplicity upper bound of 1, `get()` returns the value of the property. If the property has multiplicity upper bound > 1, `get()` returns a collection containing the values of the property. If there are no values, the collection is empty. If the property that is passed as argument is not a member of the `Class` of the `Object` instance, an exception is thrown. For example, `person.get(nameProperty) = "Person"`.

`isSet(property: Property): Boolean.` If the parameter property has multiplicity upper bound of 1, `isSet()` returns true if the value of the property is different from the default value of that property. If the parameter property has multiplicity upper bound > 1, `isSet()` returns true if the number of objects in the list is > 0. If the parameter property is not a member of the class that is obtained from the `getMetaClass()` method, an exception is thrown.

`unset(property:Property).` If the parameter property has multiplicity upper bound of 1, `unset()` atomically sets the value of the property to its default value for value-typed properties and null for object-typed properties. If the property has multiplicity upper bound > 1, `unset()` clears the collection of values of the Property. After `unset()` is called, `object.isSet(property) = false`. If the parameter property is not a member of the class that is obtained from `getMetaClass()`, an exception is thrown.

8.1.1 Discussion on the MOF Reflective Facilities

Although OCL and MOF are aligned in their respective versions 2.0, the reflective facilities of the MOF framework cannot be used in OCL expressions. The MOF reflection facilities are provided by means of the operations of the Object object type, which are defined with an informal object-oriented programming style in the MOF standard. The object types, and their features (properties and operations), that are defined in a metamodel definition $\widetilde{\mathcal{M}}$ can be referred to in an OCL expression. These object types specialize the object type Object, inheriting its operations. However, these operations cannot be used in OCL expressions because the OCL language is declarative and does not provide support for exceptions.

Enabling the use of Object operations within OCL constraints provides support to query the metamodel definition $\widetilde{\mathcal{M}}$, when an OCL constraint is evaluated over a model definition $\widetilde{M} : \mathcal{M}$, so that much more generic constraints can be defined. In this section, we discuss some issues that must be taken into account in order to enable the use of the reflective facilities within OCL expressions.

Element object type

The Element type that is defined in the metamodel definition $\widetilde{\mathrm{MOF}}$ is similar to the OCL special type OclAny, in the sense that it is the type of both objects and primitive type values. However, the Element type does not provide any operation, and is used to define the operations of the Object operation in a generic way. To provide the algebraic semantics of this type, we find the same problem that we found for the OclAny type in Section 7: it collapses the hierarchy of sorts that represent types in a theory, producing name collisions for ad-hoc overloaded operators. Therefore, we do not take this object type into account. In other object-oriented implementations of the MOF framework, like the *Eclipse Modeling Framework*, the supertype of all other object types in the metamodel definition $\widetilde{\mathrm{MOF}}$ is the Object type instead of the Element type, and primitive types are defined independently. We also follow this approach.

ReflectiveCollection and ReflectiveSequence object types

These object types permit using collections of elements to manipulate an object by means of Object operations. However, they are not so expressive as the collection types of the OCL language, because they do not take the uniqueness feature into account . Furthermore, they constitute an alternative to OCL collection types that makes the infrastructure of object types more complex. To achieve a complete alignment with the OCL specification, we reuse the OCL collection types to deal with collections of either primitive type values or Object instances.

Null values

Object operations may either return a null value or throw an exception when the corresponding operation is not defined for the argument values. The OCL language is declarative and does not provide support for exceptions. In order to align OCL and MOF, these error cases are considered as undefined values so that they can be used in OCL expressions.

Only side-effect free operators

The OCL language is side-effect free, so that only the operators that do not change the property values of an object are allowed in OCL expressions: *getMetaClass*, *container*, *get*, and *isSet*. However, we also provide the semantics of the *set* and *unset* operations, in order to support model management. In addition, the *equals* operator is replaced by the OCL operator `=`.

In the following section, we define the semantics of the Object object type and its operations. Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the Object operations that do not have side-effects can be used to define terms that represent OCL expressions in a theory $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$.

8.2 Semantics of the MOF Reflection Facilities

Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the object types that are defined in $\widetilde{\mathcal{M}}$ specialize the MOF Object object type, inheriting its operations. This specialization relationship is always defined implicitly and never appears defined in $\widetilde{\mathcal{M}}$. There is a subtheory, called `META-MODEL`, that is defined as subtheory of the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory. The theory `META-MODEL` provides the sort *MetaObject*, whose terms represent Object instances, and the sort *ModelType*{*MetaObject*}, whose terms represent model definitions that are constituted of Object instances. A term of sort *MetaObject* is defined by means of a constructor that is common to all metamodel specification realizations. This fact permits defining objects in model definitions $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, when the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory is not provided.

$$\begin{tikzcd}
\text{untyped level} & \widehat{o} \arrow[r, mapsto, "\widehat{\psi}"] & \widehat{o}' \arrow[d, mapsto, "downObject"] \\
\text{typed level} & \widetilde{o} \arrow[u, mapsto, "upObject"] \arrow[r, mapsto, "\widetilde{\psi}"] & \widetilde{o}'
\end{tikzcd}$$

Figure 8.3: Manipulation of an object by means of the MOF Reflective Facilities.

$$\begin{tikzcd}
\text{untyped level} & \widehat{M} \arrow[r, mapsto, "\widehat{\phi}"] & \widehat{M}' \arrow[d, mapsto, "downModel"] \\
\text{typed level} & \widetilde{M} \arrow[u, mapsto, "upModel"] \arrow[r, mapsto, "\widetilde{\phi}"] & \widetilde{M}'
\end{tikzcd}$$

Figure 8.4: Manipulation of a model definition by means of the MOF Reflective Facilities.

Given an object type OT, such that $\widetilde{\mathrm{OT}} : \mathrm{MOF}$, $root(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}) : \textsc{Class}$ and $root(\widetilde{\mathrm{OT}}, \widetilde{\mathrm{MOF}}) \subseteq root(\widetilde{\mathcal{M}}, \widetilde{\mathrm{MOF}})$, an object $\widetilde{o}$, such that $\widetilde{o} : \mathrm{OT}$, that belongs to a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, can be queried and manipulated by means of Object operations. However, $\widetilde{o}$ has to be represented first as an Object instance, denoted by $\widehat{o}$. This syntactical representation change is obtained by means of the equationally-defined functions

$$upObject : Object\#\mathcal{M} \longrightarrow MetaObject$$

$$downObject : MetaObject \times ModelType\{\mathrm{MOF}\} \rightsquigarrow Object\#\mathcal{M}$$

so that $downObject(upObject(\widetilde{o}), \widetilde{\mathcal{M}}) = \widetilde{o}$ and $upObject(downObject(\widehat{o}, \widetilde{\mathcal{M}})) = \widehat{o}$. Therefore, an equationally-defined function $\widetilde{\psi} : Object\#\mathcal{M} \longrightarrow Object\#\mathcal{M}$ that manipulates an object $\widetilde{o}$ can be defined, by using the MOF reflection facilities, as a composite function $\widetilde{\phi} = downObject \circ \widehat{\psi} \circ upObject$, where the equationally-defined function $\widehat{\psi} : MetaObject \longrightarrow MetaObject$ manipulates the metarepresentation $\widehat{o}$ of an object $\widetilde{o}$ by means of Object operations. This process is illustrated in Fig. 8.3.

At a coarser level of granularity, a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, can be manipulated by means of an equationally-defined function $\widetilde{\phi} : ModelType\{\mathcal{M}\} \longrightarrow ModelType\{\mathcal{M}\}$, which constitutes a model transformation. The $\widetilde{\phi}$ function manipulates the objects that are defined in $\widetilde{M}$, so that Object operations may also be used to manipulate them. To enable the application of Object operations to the objects $\widetilde{o}$ that constitute $\widetilde{M}$, the objects $\widetilde{o}$ have to be represented as Object instances $\widehat{o}$. The *upObject* and *downObject* functions are extended to consider model definitions as arguments, by means of the equationally-defined functions

$$upModel : ModelType\{\mathcal{M}\} \longrightarrow ModelType\{MetaObject\}$$

$$downModel : ModelType\{MetaObject\} \times ModelType\{\mathrm{MOF}\} \rightsquigarrow ModelType\{\mathcal{M}\},$$

where $downModel(upModel(\widetilde{M}), \widetilde{\mathcal{M}}) = \widetilde{M}$ and $upModel(downModel(\widehat{M}, \widetilde{\mathcal{M}})) = \widehat{M}$. Therefore, the function $\widetilde{\phi}$ can be defined as a composite function $\widetilde{\phi} = downModel \circ \widehat{\phi} \circ upModel$, where the equationally-defined function $\widehat{\phi} : ModelType\{MetaObject\} \longrightarrow ModelType\{MetaObject\}$ performs the model transformation at the untyped metalevel. This process is illustrated in Fig. 8.4.

The `META-MODEL` theory, together with its subtheory inclusion in each theory $reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, formally defines the notion of MOF Reflection Facilities, that permit the manipulation of the metarepresentation of model definitions $\widetilde{M}$. This theory introduces an extra sublevel, in the conceptual levels M1, M2, and M3 of the MOF Framework, where the metarepresentation $\widehat{M}$ of a model definition $\widetilde{M}$ is performed in a type-agnostic way, i.e., without the corresponding metamodel specification realization $reflect(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. We call *typed level* to the metalevel where objects are defined as objects $\widetilde{o}$ of specific object types, and we call *untyped level* to the metalevel where objects are defined as Object instances in an inde-

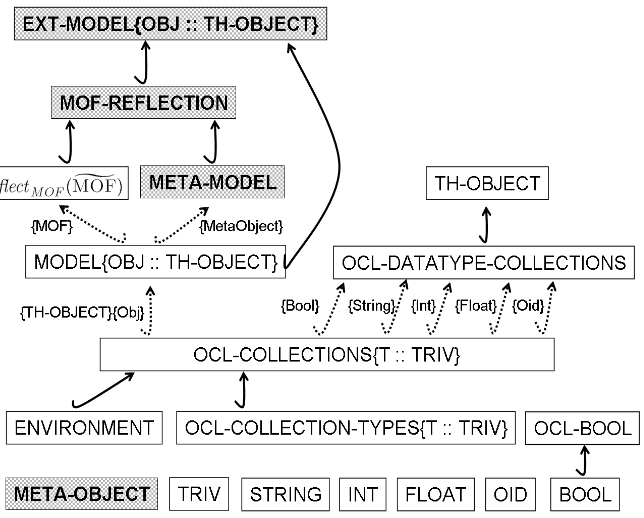


Figure 8.5: The MOF Reflection API

pendent way of the corresponding object type. Since metamodel definitions $\widetilde{\mathcal{M}}$ are also model definitions $\widetilde{M}$, such that $\widetilde{M}$: MOF, metamodel definitions can also be manipulated in a type-agnostic way. In subsequent sections, (1) the `META-MODEL` theory is presented, completing the infrastructure of theories that has been developed throughout the Sections 6 and 7; (2) the semantics of the Object type is provided; and (3) a mathematical definition of the Object operations is given.

8.2.1 The `META-MODEL` theory

The `META-MODEL` theory permits representing model defintions $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, in a type-agnostic way; i.e., without taking the metamodel realization $reflect_{MOF}(\widetilde{\mathcal{M}})$ into account. This theory is defined by the equation

$$\texttt{META-MODEL} = \texttt{MODEL}\{\texttt{MetaObject}\},$$

where `MetaObject` is a view that maps the `TH-OBJECT` theory to the `META-OBJECT` theory. The `META-OBJECT` theory provides the sort *MetaObject*, whose terms represent Object instances. The `META-OBJECT` and `META-MODEL` theories complete the theory infrastructure of our MOF framework, as illustrated in Fig. 8.5.

The metarepresentation of any object $\widetilde{o}$, which is defined at the typed metalevel, as a MOF Object instance at the untyped metalevel involves the metarepresentation of all of the values that can participate in the definition of $\widetilde{o}$: primitive type values, enumeration literals, object identifiers, OCL collections of the previous values, and the object itself. On the one hand, values of the OCL predefined types, such as the primitive types and the OCL collection types, are metarepresented by means of the identity function. For example, the value 1 that may be the property value of an object $\widetilde{o}$, at the typed metalevel, is also represented as the value 1, at the MOF metalevel. On the other hand, values of user-defined types have a specific metarepresentation at the untyped metalevel. Enumeration literals and object identifiers are metarepresented by taking into account the following sorts and operators:

- *Enumeration Literals* are metarepresented as terms of the sort `MetaEnum`, whose constructor is defined in the following mel theory, shown in Maude notation as:

```
fmod META-ENUM is
    protecting STRING .
    sort MetaEnum .
    op metaEEnumLiteral : String -> MetaEnum .
    op nullMetaEEnumLiteral : -> [MetaEnum] .
endfm
```

The literal `rdbms/RDataType/VARCHAR` of the `rdbms/RDataType` enumeration type in the RDBMS metamodel is meta-represented as

```
metaEEnumLiteral("rdbms/RDataType/VARCHAR").
```

- Object identifiers are user-dependent, since their constructor is defined from a Class instance, as shown in Section 6. Identifiers are metarepresented as terms of sort `MetaOid`, which is defined in the following theory, in Maude notation,

  ```
  fmod META-OID is
      protecting QID .
      protecting STRING .

      sort MetaOid .
      op metaOid : String Qid -> MetaOid .
      op nullMetaOid : -> [MetaOid] .
  endfm
  ```

  An identifier `oid#Class('0)` is metarepresented as `metaOid("oid#Class", '0)`.

These theories only permit metarepresenting enumeration literals and object identifiers at the MOF metalevel. However, collections of literals or object identifiers cannot be metarepresented yet. To permit their metarepresentation, we define the `META-OBJECT` theory, which instantiates the `OCL-COLLECTIONS{X :: TRIV}` theory with the views *MetaEnum* and *MetaOid*. These views map the TRIV theory to the corresponding theories as follows:

```
view MetaEnum from TRIV to META-ENUM is
    sort Elt to MetaEnum .
endv

view MetaOid from TRIV to META-OID is
    sort Elt to MetaOid .
endv
```

Therefore, in the `META-MODEL` theory, collections of literals and collections of object identifiers can also be metarepresented. In addition, the `META-MODEL` theory permits meta-representing object properties and objects. Object properties are metarepresented as terms of the sorts `MetaProperty` and `MetaPropertySet` by means of the following set of sorts and constructors:

```
sorts MetaProperty MetaPropertySet .
subsort MetaProperty < MetaPropertySet .

op _`,_ : MetaPropertySet MetaPropertySet -> MetaPropertySet
    [assoc comm id: noneMetaProperty] .
op noneMetaProperty : -> MetaPropertySet .
```

Depending on the type of property, specific object properties are metarepresented as follows:

- *Set properties:* Properties that are initialized with a specific value are metarepresented by means of the following constructors:

  ```
  op property`:_=_ : String Element{String} -> MetaProperty  .
  op property`:_=_ : String Element{Int} -> MetaProperty  .
  op property`:_=_ : String Element{Float} -> MetaProperty  .
  op property`:_=_ : String Element{Bool} -> MetaProperty  .
  op property`:_=_ : String Element{MetaOid} -> MetaProperty  .
  op property`:_=_ : String Element{MetaEnum} -> MetaProperty  .
  ```

  For example, the property `name : "Person"` is metarepresented as `property : "name" = "Person"`.

- *Unset properties:* Properties that have not been set with any value yet are metarepresented by means of the constructor:

```
op property`:_ : String -> MetaProperty .
```

For example, the property `name` is metarepresented as `property : "name"`.

- *Object type name:* a special property is defined to metarepresent the object type name of a MOF *Object* instance:

```
op class`:_ : String -> MetaProperty .
```

For example, the constant `rdbms/Table` is metarepresented as `class : "rdbms/Table"`.

The MOF Object object type is represented by means of the following sorts and constructors in the `META-OBJECT` theory:

```
sorts MetaObject MetaCid .
op MOF/Object : -> MetaCid .
op <_:`MOF/Object`|_> : MetaOid MetaPropertySet -> MetaObject .
```

For example, the `Table` instance of the relational schema of the example

```
< oid#Table('Table0) : Table |
    name : "Person",
    column : OrderedSet{ oid#Column('Column0)
        :: oid#Column('Column1)
        :: oid#Column('Column2) },
    primaryKey : OrderedSet{oid#PrimaryKey('PK0)},
    foreignKey,
    schema : oid#Schema('Schema0)
>
```

is metarepresented as a MOF Object instance as follows:

```
< metaOid("oid#Table", 'Table0) : Object |
    class : "Table",
    property : "name" = "Person",
    property : "column" =
        OrderedSet{ metaOid("oid#Column", 'Column0)
        :: metaOid("oid#Column", 'Column1)
        :: metaOid("oid#Column", 'Column2) },
    property : "primaryKey" =
        OrderedSet{metaOid("oid#PrimaryKey",'PK0)},
    property : "foreignKey",
    property : "schema" = metaOid("oid#Schema", 'Schema0)
>
```

We refer to terms of the sort *MetaObject* as MOF Object instances. This fact is indicated by means of the `MOF/Object` object type name, which forms part of the constructor symbol for metaobjects. Terms of sort *MetaObject* can be viewed as the data metarepresentation of the objects of a model definition $\widetilde{M}$. Several projector operators are also provided to obtain the different subterms that constitute a *MOFObject* term.

```
op oid : MetaObject -> MetaOid .
eq oid( < MOID:MetaOid : MOF/Object | MPS:MetaPropertySet > ) =
    MOID:MetaOid .

op class : MetaObject -> MetaCid .
eq class( < MOID:MetaOid : MOF/Object | MPS:MetaPropertySet > ) =
    MOF/Object .

op getProperties : MetaObject -> MetaPropertySet .
eq getProperties( < MOID:MetaOid : MOF/Object | MPS:MetaPropertySet > ) =
    MPS:MetaPropertySet > .
```

A metarepresented object $\widehat{o}$ is defined in a way independent of its corresponding metamodel realization $reflect_{\text{MOF}}(\widetilde{\mathcal{M}})$. However, the definition, as data, of the corresponding object type is still needed to check if the metarepresented object is well-formed. A metarepresented object contains the name of its object type, called meta object type, by means of the `class property`. We define a new operator that projects the name of the meta object type as:

```
op metaClassName : MetaEObject -> String .
eq class( < MOID:MetaOid : ecore/Object |
    class : name:String, MPS:MetaPropertySet > ) =
    name:String .
```

A model definition $\widetilde{M}$ can be represented as a configuration $\widehat{M}$ of MOF Object instances. The `META-MODEL` theory instantiates the `MODEL{OBJ :: TH-OBJECT}` theory by means of the expression `MODEL{MetaObject}`, where the view `MetaObject` is defined as follows

```
view MetaObject from TH-EOBJECT to META-OBJECT is
    sort Cid to MetaCid .
    sort Object to MetaObject .
    sort ObjectOid to MetaOid .
    sort Property to MetaProperty .
    sort PropertySet to MetaPropertySet .
    op noneProperty to noneMetaProperty .
    op nullObject to nullMetaObject .
endv
```

Therefore, the `META-MODEL` theory reuses the support for OCL, which is provided in the `OCL-COLLECTION-TYPES{T :: TRIV}` and `OCL-COLLECTIONS{T :: TRIV}`, and the support for both defining collections of objects and navigating through object-typed properties, which is provided in the `MODEL{OBJ :: TH-OBJECT}` theory.

Let $\widetilde{o}$ be an object, at the typed metalevel, in a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF, or in a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, and let $\widehat{o}$ be its metarepresentation at the typed metalevel. The mappings $\widetilde{o} \mapsto \widehat{o}$ and $\widehat{o} \mapsto \widetilde{o}$ are provided by two equationally-defined functions that are declared as follows:

$$upObject : Object\#\mathcal{M} \rightarrow MetaObject$$

$$downObject : MetaObject \times ModelType\{\text{MOF}\} \rightsquigarrow Object\#\mathcal{M}$$

The function *upModel* is defined as the extension of the *upObject* function by applying it to each object in a model definition $\widetilde{M}$. Likewise, the function *downModel* is defined as the extension of the *downObject* function by applying it to each MOF Object instance in a model $\widehat{M}$, such that $\widehat{M} = upModel(\widetilde{M})$.

The MOF meta-metamodel realization is provided by the $reflect_{\text{MOF}}(\widetilde{\text{MOF}})$ theory, in and adhoc way, due to its self-recursive definition. The functions *upObject/downObject* and *upModel/downModel* are defined for the MOF meta-metamodel in the `MOF-REFLECTION` theory, as follows:

```
    op upObject : Object#MOF -> MetaObject .
    op downObject : MetaObject ModelType{MOF} [Object#MOF] ~>
        Object#MOF .

    op upModel : ModelType{MOF} -> ModelType{MetaObject} .
    op downModel : ModelType{MetaObject} ModelType{MOF} [Object#MOF] ~>
        ModelType{MOF} .
```

where the last argument of the `downObject` and `downModel` operators is a value of the kind of the corresponding `Object#`$\mathcal{M}$ sort, so that the operator symbol `downModel` can be used for other metamodel realizations, as shown below. For example, the term `downModel( upModel(` $\widetilde{\text{MOF}}$ `),` $\widetilde{\text{MOF}}$ `, nullObject#MOF)` is reduced to $\widetilde{\text{MOF}}$. For any other metamodel, this functions are likewise defined generically in the `EXT-MODEL{OBJ :: TH-OBJECT}` theory, as follows:

```
    op upObject : Object#MOF -> MetaObject .
```

```
op downObject : MetaObject ModelType{MOF} [Object#OBJ] ~>
    Object#MOF .

op upModel : ModelType{MOF} -> ModelType{MetaObject} .
op downObject : ModelType{MetaObject} ModelType{MOF} [Object#OBJ] ~>
    ModelType{OBJ} .
```

8.2.2 Semantics of the MOF Object object type

In our framework, the semantics of the MOF Object object type is defined in the **META-MODEL** theory, independently of any metamodel realization $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$, by means of the equation

$$\boxed{[\![\textsc{Object}]\!]_{\mathrm{MOF}} = T_{\mathtt{META\text{-}MODEL},\mathit{MetaObject}}}$$

so that only metarepresentations $\widehat{o}$, at the untyped metalevel, of objects $\widetilde{o}$, at the typed metalevel, are instances of the Object object type. The *instanceOf* relation is defined for the Object object type by the equivalence

$$\boxed{\widehat{o} : \textsc{Object} \iff \widehat{o} \in [\![\textsc{Object}]\!]_{\mathrm{MOF}}}.$$

The **META-MODEL** theory introduces three sorts to define metarepresented model definitions $\widehat{M}$: *Configuration{MetaObject}*, *ModelType{MetaObject}*, and *ConsistentModelType{MetaObject}*. Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, and its realization $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, which is a theory that includes the **META-MODEL** theory as shown in Fig. 8.5, the sorts that are provided in the **META-MODEL** theory are:

- *Configuration{MetaModel}*, also denoted by ${\mathcal{M}_0}^{\vartriangle}$ for short, whose terms represent collections of metarepresented objects. The semantics of the ${\mathcal{M}_0}^{\vartriangle}$ sort is defined as follows:

$$\boxed{\begin{aligned}[\![{\mathcal{M}_0}^{\vartriangle}]\!]_{\mathrm{MOF}} = \{\widehat{M} \mid \widehat{M} \in T_{\mathtt{META\text{-}MODEL},\mathit{Configuration\{MetaObject\}}} \,\wedge \\ \mathit{downModel}(\widehat{M}, \widetilde{\mathcal{M}}) : \mathcal{M}_0\}.\end{aligned}}$$

  and the *isTypeOf* relation $\widetilde{M} : \mathcal{M}_0$ is metarepresented by the equivalence

$$\boxed{\widehat{M} : {\mathcal{M}_0}^{\vartriangle} \iff \widehat{M} \in [\![{\mathcal{M}_0}^{\vartriangle}]\!]_{\mathrm{MOF}}}.$$

- *ModelType{MetaModel}*, also denoted by $\mathcal{M}^{\vartriangle}$ for short, whose terms represent metarepresented model definitions $\widetilde{M}$. The semantics of the $\mathcal{M}^{\vartriangle}$ sort is defined as follows:

$$\boxed{\begin{aligned}[\![\mathcal{M}^{\vartriangle}]\!]_{\mathrm{MOF}} = \{\widehat{M} \mid \widehat{M} \in T_{\mathtt{META\text{-}MODEL},\mathit{ModelType\{MetaObject\}}} \,\wedge \\ \widehat{M} : {\mathcal{M}_0}^{\vartriangle} \;\wedge\; \mathit{downModel}(\widehat{M}, \widetilde{\mathcal{M}}) : \mathcal{M}\}\end{aligned}}$$

  and the *structural conformance* relation $\widetilde{M} : \mathcal{M}$ is metarepresented by the equivalence

$$\boxed{\widehat{M} : \mathcal{M}^{\vartriangle} \iff \widehat{M} \in [\![\mathcal{M}^{\vartriangle}]\!]_{\mathrm{MOF}}}.$$

- *ConsistentModelType{MetaModel}*, also denoted by $(\mathcal{M}, \mathcal{C})^{\vartriangle}$ for short, whose terms represent metarepresented model definitions $\widetilde{M}$ that satisfy a set $\mathcal{C}$ of OCL constraints. The semantics of the $\mathcal{M}^{\vartriangle}$ sort is defined as follows:

$$\boxed{\begin{aligned}[\![(\mathcal{M}, \mathcal{C})^{\vartriangle}]\!]_{\mathrm{MOF}} = \{\widehat{M} \mid \widehat{M} \in T_{\mathtt{META\text{-}MODEL},\mathit{ConsistentModelType\{MetaObject\}}} \,\wedge \\ \widehat{M} : \mathcal{M}^{\vartriangle} \;\wedge\; \mathit{downModel}(\widehat{M}, \widetilde{\mathcal{M}}) : (\mathcal{M}, \mathcal{C})\}\end{aligned}}$$

  and the *constrained conformance* relation $\widetilde{M} : (\mathcal{M}, \mathcal{C})$ is metarepresented by the equivalence

$$\boxed{\widehat{M} : (\mathcal{M}, \mathcal{C})^{\vartriangle} \iff \widehat{M} \in [\![(\mathcal{M}, \mathcal{C})^{\vartriangle}]\!]_{\mathrm{MOF}}}.$$

Given a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, the tree and graph structure of the metarepresentation $\widehat{M}$ of a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, is checked by means of the corresponding metamodel realization $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$. A metarepresentation $\widehat{M}$ of a model definition $\widetilde{M}$ satisfies the set $\mathcal{C}$ of OCL constraints iff $\widetilde{M} : (\mathcal{M}, \mathcal{C})$, as defined in the metamodel specification realization $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$. However, the structural conformance relation $\widehat{M} : \mathcal{M}^{\triangle}$, and the constrained conformance relation $\widetilde{M} : (\mathcal{M}, \mathcal{C})$, can also be metarepresented at the MOF metalevel, so that the theories $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ and $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ are not needed. In the following sections we explain: (i) the semantics of the MOF Object operations, (ii) how they can be used to traverse the graph and the tree structure of a metarepresented model definition $\widehat{M}$, such that $\widehat{M} : \mathcal{M}^{\triangle}$, in a generic way, and (iii) how OCL constraints can also be metarepresented at the MOF metalevel, providing support for the structural conformance relation $\widehat{M} : \mathcal{M}^{\triangle}$ and the constrained conformance relation $\widetilde{M} : (\mathcal{M}, \mathcal{C})$ at the MOF metalevel.

8.2.3 Semantics of the MOF Object Operations

Given a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF, the MOF Object object type provides a set of operations that permits querying and manipulating the properties of metarepresented objects $\widehat{o}$, such that $\widehat{o}$: Object, that belong to a metarepresented model definition $\widehat{M}$, such that $\widehat{M} : \mathcal{M}^{\triangle}$. In this section, we explain how these operations are specified as equationally-defined functions in the `META-MODEL` theory:

- *get:* This operation obtains the value of a metarepresented property. The *get* operation is defined as two sets of equationally-defined functions in the `META-MODEL` theory, depending on the type of the returning value:

  – *Returns a basic data type value or a collection of basic values:* Given the set

  $$T = \{\mathit{Bool}, \mathit{String}, \mathit{Int}, \mathit{Float}, \mathit{MetaEnum}, \mathit{MetaOid}\}$$

  of sort names in the theory `META-MODEL`, we define a family of indexed functions

  $$\{\mathit{get}_t : [\![\text{Object}]\!]_{\mathrm{MOF}} \times [\![\text{String}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}^{\triangle}]\!]_{\mathrm{MOF}} \rightsquigarrow T_{\texttt{META-MODEL}, \mathit{Collection}+\{t\}}\}_{t \,\in\, T},$$

  where each function get_t obtains a value of sort $\mathit{Collection}+\{t\}$ when the metarepresented property is typed with the corresponding type t. This partial function is defined for tuples of the form $(\widehat{o}, \mathit{name}, \widehat{\mathcal{M}})$, where $\widehat{o}$: Object, name : String, and $\widehat{\mathcal{M}} : \mathrm{MOF}^{\triangle}$. In this case, the get_t function is defined by means the following equalities:

  $$\mathit{get}_t(\widehat{o}, \mathit{name}, \widehat{\mathcal{M}}) = \mathit{value} \text{ such that } \mathit{value} \in T_{\texttt{META-MODEL}, \mathit{Collection}+\{t\}}$$
  $$\text{when } (\mathit{property} : \mathit{name} = \mathit{value}) \in \mathit{getProperties}(\widehat{o})$$

  $$\mathit{get}_t(\widehat{o}, \mathit{name}, \widehat{\mathcal{M}}) = \mathit{defaultValue}_t(\widehat{o}, \mathit{name}, \widehat{\mathcal{M}})$$
  $$\text{when } (\mathit{property} : \mathit{name}) \in \mathit{getProperties}(\widehat{o})$$

  where each function in the family of indexed functions

  $$\{\mathit{defaultValue}_t : [\![\text{Object}]\!]_{\mathrm{MOF}} \times [\![\text{String}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}^{\triangle}]\!]_{\mathrm{MOF}} \rightsquigarrow T_{\texttt{META-MODEL}, \mathit{Collection}\{t\}}\}_{t \in T},$$

  obtains the default value for a metarepresented property, depending on its definition in $\widehat{\mathcal{M}}$. A property of a metarepresented object $\widehat{o}$ is defined by means of a Property instance $\widetilde{p}$ that is also metarepresented in $\widehat{\mathcal{M}}$. A Property instance $\widetilde{p}$ is defined by means of multiplicity meta-properties (*lower*, *upper*, *ordered*, *unique*), and the *defaultValue* and *type* meta-properties, as indicated in Section 6. A $\mathit{defaultValue}_t$ function is only defined when the type of the metarepresented property is neither a collection nor an object type, and its *type* meta-property corresponds to t. In addition, the *name* meta-property of $\widetilde{p}$ must coincide with the *name* argument. When the meta-property *defaultValue* of $\widetilde{p}$ is set, the returned value is the *defaultValue* meta-property value. In any other case, the $\mathit{defaultValue}_t$ is defined as follows:

$$\begin{aligned}
&defaultValue_{Bool}(\widehat{o}, name, \widehat{\mathcal{M}}) = false\\
&defaultValue_{String}(\widehat{o}, name, \widehat{\mathcal{M}}) = ""\\
&defaultValue_{Int}(\widehat{o}, name, \widehat{\mathcal{M}}) = 0\\
&defaultValue_{Float}(\widehat{o}, name, \widehat{\mathcal{M}}) = 0.0
\end{aligned}$$

These functions are provided as equationally-defined operators in the `META-MODEL` theory. These operators are declared, in Maude notation, as follows:

```
op _.‘getBool‘(_,_‘) : MetaObject String
    ModelType{MetaObject} ~> Collection+{Bool} .
op _.‘getString‘(_,_‘) : MetaObject String
    ModelType{MetaObject} ~> Collection+{String} .
op _.‘getInt‘(_,_‘) : MetaObject String
    ModelType{MetaObject} ~> Collection+{Int} .
op _.‘getFloat‘(_,_‘) : MetaObject String
    ModelType{MetaObject} ~> Collection+{Float} .
op _.‘getMetaEnum‘(_,_‘) : MetaObject String
    ModelType{MetaObject} ~> Collection+{MetaEnum} .
op _.‘getMetaOid‘(_,_‘) : MetaObject String
    ModelType{MetaObject} ~> Collection+{MetaOid} .
```

These functions metarepresent, at the untyped metalevel, the projector functions that permit obtaining property values from an object $\widetilde{o}$, such that $\widetilde{o} \in T_{reflect_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),Object\#\mathcal{M}}$, and $\widetilde{o} \in \widetilde{M}$ where $\widetilde{M} : \mathcal{M}$, at the typed metalevel. For example, given an object $\widetilde{t}$ that represents the following Table instance

```
< oid#Table('Table0) : Table |
    name : "Person",
    column : OrderedSet{ oid#Column('Column0)
        :: oid#Column('Column1)
        :: oid#Column('Column2) },
    primaryKey : OrderedSet{oid#PrimaryKey('PK0)},
    foreignKey,
    schema : oid#Schema('Schema0)
>
```

in a model definition $\widetilde{M}$, such that $\widetilde{M}$: RDBMS. The OCL expression $\widetilde{t}$. `name` is metarepresented at the MOF metalevel as

$$\texttt{upObject}(\widetilde{t})\ .\ \ \texttt{getString("name", upModel(}\widetilde{\mathrm{RDBMS}}\texttt{))}.$$

Note that the last argument $\widetilde{\mathrm{RDBMS}}$ is needed to compute the default value of the property in case the corresponding value-typed property is *unset*.

– *Returns a metarepresented object or a collection of metarepresented objects*: When the metarepresented property is typed with an object type and is set with a collection of metarepresented object identifiers, the $get_{MetaObject}$ operation permits the navigation of the graph structure that is kept in a metarepresented model definition $\widehat{M}$. The function

$$get_{MetaObject} : [\![\mathrm{OBJECT}]\!]_{\mathrm{MOF}} \times [\![\mathrm{STRING}]\!]_{\mathrm{MOF}} \times [\![\mathcal{M}^{\triangle}]\!]_{\mathrm{MOF}} \rightsquigarrow T_{\texttt{META-MODEL},Collection\{MetaObject\}},$$

permits navigating through metarepresented object-typed properties. This function is provided as an equationally-defined function in the `META-MODEL` theory as:

```
op _.‘getMetaObject‘(_,_‘) :
    MetaObject String ModelType{MetaObject}
    ~> Collection+{MetaObject}
```

This function is defined for tuples of the form $(\widehat{o}, \mathit{name}, \widehat{M})$, where $\widehat{o} : \textsc{Object}$, $\mathit{name} : \textsc{String}$, and $\widehat{M} : \mathcal{M}^{\triangle}$. The $get_{MetaObject}$ operator metarepresents, at the untyped metalevel, the projector function that permits navigating the graph structure of a model definition through object-typed properties at the typed metalevel. For the example given above, the OCL expression that is represented by the term

$$\widetilde{t}\ \texttt{.}\ \ \texttt{schema(}\ \widetilde{M}\ \texttt{)}$$

at the MOF base level, is metarepresented as the term

$$\texttt{upObject(}\ \widetilde{t}\ \texttt{)}\ \ \texttt{.}\ \ \texttt{getMetaObject("schema", upModel(}\ \widetilde{M}\ \texttt{))}$$

at the untyped metalevel.

- *set:* The *set* operation is used to assign a value to an object property. The *set* operation is defined as a family of functions that are indexed by the type of the corresponding property to be initialized:

$$\{set_t : [\![\textsc{Object}]\!]_{\text{MOF}} \times [\![\textsc{String}]\!]_{\text{MOF}} \times T_{\texttt{META-MODEL},Collection+\{t\}} \rightsquigarrow [\![\textsc{Object}]\!]_{\text{MOF}}\}_{t\,\in\,T}.$$

A function set_t is defined for tuples of the form $(\widehat{o}, \mathit{name}, \mathit{value})$, where: $\widehat{o}$, such that $\widehat{o} : \textsc{Object}$, is the metarepresented object that owns the property to be initialized; *name*, such that $\mathit{name} : \textsc{String}$, is the name of the property to be initialized; and *value*, such that $\mathit{value} \in T_{\texttt{META-MODEL},Collection+\{t\}}$, is the new value for the property. To define a function set_t, we find two cases:

  - When the property to be initialized is still *unset*, i.e., $(\mathit{property} : \mathit{name}) \in \mathit{getProperties}(\widehat{o})$, a set_t function is defined by the equality:

$$set_t(\widehat{o}, \mathit{name}, \mathit{value}) = \widehat{o'},$$

  where $(\mathit{property} : \mathit{name} = \mathit{value}) \in \mathit{getProperties}(\widehat{o'})$ and $(\mathit{property} : \mathit{name}) \notin \mathit{getProperties}(\widehat{o'})$.

  - When the property to be initialized is already *set*, i.e., $(\mathit{property} : \mathit{name} = \mathit{oldValue}) \in \mathit{getProperties}(\widehat{o})$, where $\mathit{oldValue} \in T_{\texttt{META-MODEL},Collection+\{t\}}$, a set_t function is defined by the equality:

$$set_t(\widehat{o}, \mathit{name}, \mathit{value}) = \widehat{o'},$$

  where $(\mathit{property} : \mathit{name} = \mathit{value}) \in \mathit{getProperties}(\widehat{o'})$ and $(\mathit{property} : \mathit{name} = \mathit{oldValue}) \notin \mathit{getProperties}(\widehat{o'})$.

These functions are specified as equationally-defined operators in the `META-MODEL` theory, in Maude notation, as follows:

```
op _.`set`(_`,_`) : MetaObject String Collection+{Bool}
    ~> MetaObject .
op _.`set`(_`,_`) : MetaObject String Collection+{String}
    ~> MetaObject .
op _.`set`(_`,_`) : MetaObject String Collection+{Int}
    ~> MetaObject .
op _.`set`(_`,_`) : MetaObject String Collection+{Float}
    ~> MetaObject .
op _.`set`(_`,_`) : MetaObject String Collection+{MetaEnum}
    ~> MetaObject .
op _.`set`(_`,_`) : MetaObject String Collection+{MetaOid}
    ~> MetaObject .
op _.`set`(_`,_`) : MetaObject String Collection+{MetaObject}
    ~> MetaObject .
```

The last operator permits initializing an object-typed property with the collection of object identifiers that correspond to a given collection of objects. A set_t function is not side-effect free, because it manipulates a metarepresented object. A function of this kind does not metarepresent any OCL operator but provides the capability to manipulate model definitions by manipulating their constituent objects. For example, the property *column* of the $\textsc{Table}$ instance $\widetilde{t}$, defined above, can be manipulated by means of the following term:

$$\begin{array}{l}\texttt{upObject(}\ \widetilde{t}\ \texttt{).set("column",}\\ \quad\texttt{OrderedSet\{ metaOid("oid\#Column", "Column0") \}) .}\end{array}$$

- *getMetaClass:* The function

$$getMetaClass : [\![\textsc{Object}]\!]_{\text{MOF}} \times [\![\text{MOF}^{\triangle}]\!]_{\text{MOF}} \rightsquigarrow [\![\textsc{Object}]\!]_{\text{MOF}}.$$

is a partial function that provides metainformation about the object type of a given metarepresented object $\widehat{o}$. Recall that given an object $\widetilde{o}$, such that $\widetilde{o} \in T_{reflect_{\text{MOF}}(\widetilde{\mathcal{M}}),Object\#\mathcal{M}}$, at the typed metalevel, the $\textsc{Object}$ instance $\widehat{o}$ metarepresents $\widetilde{o}$ at the untyped metalevel by means of the *upObject* function, i.e., $\widehat{o} = upObject(\widetilde{o})$. The *getMetaClass* function obtains the metarepresentation $\widehat{cl}$ of the $\textsc{Class}$ instance $\widetilde{cl}$, such that $\widetilde{cl} : \textsc{Class}$ and $\widetilde{cl} \in \widetilde{\mathcal{M}}$, that forms part of the object type defintion $\widetilde{\text{OT}}$, such that $\widetilde{o}$: OT. That is, there is an object type definition $\widetilde{\text{OT}}$, such that $\widetilde{\text{OT}}$: MOF, $tree(\widetilde{\text{OT}}, \widetilde{\text{MOF}}) \subseteq tree(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}})$, $root(\widetilde{\text{OT}}, \widetilde{\text{MOF}}) = \widetilde{cl}$, and $\widetilde{o}$: OT. The *getMetaClass* function provides the introspection capability to a MOF framework by bridging an arbitrary metarepresented object $\widehat{o}$ to the metadata that describes its object type in the corresponding metamodel definition.

The *getMetaClass* function is defined for pairs of the form $(\widehat{o}, \widehat{\mathcal{M}})$, where $\widehat{o} : \textsc{Object}$ and $downObject(\widehat{o}) \in T_{reflect_{\text{MOF}}(\widetilde{\mathcal{M}}),Object\#\mathcal{M}}$, and $\widehat{\mathcal{M}} : \text{MOF}^{\triangle}$. In this case, the semantics of the *getMetaClass* function is defined by the equality:

$$getMetaClass(\widehat{o}, \widehat{\mathcal{M}}) = \widehat{mo},$$

such that

$$\begin{aligned}
&\widehat{mo} : \textsc{Object} \wedge \\
&\widehat{mo} \in \widehat{\mathcal{M}} \wedge \\
&downObject(\widehat{mo}) : \textsc{Class} \wedge \\
&get_{String}(\widehat{mo}, "name", \widehat{\text{MOF}}) = metaClassName(\widehat{o}).
\end{aligned}$$

This function is specified in the `META-MODEL` theory as the equationally-defined operator

```
op _.‘getMetaClass‘(_‘) : MetaObject ModelType{MetaObject}
    ~> MetaObject .
```

For example, the metarepresentation of the $\textsc{Class}$ instance that constitutes the root object of the $\widetilde{\textsc{Table}}$ object type definition in the metamodel definition $\widetilde{\text{RDBMS}}$ is obtained from the $\textsc{Table}$ instance $\widetilde{t}$ that is provided above, as follows:

$$\texttt{upObject(}\ \widetilde{t}\ \texttt{).getMetaClass(upModel(}\ \widetilde{\text{RDBMS}}\ \texttt{))}$$

Note that the *instanceOf* relation is reified as data at the untyped metalevel by means of the *class* meta-property of $\textsc{Object}$ instances. Given a model definition $\widetilde{M}$ and a metamodel definition $\widetilde{\mathcal{M}}$, this meta-property constitutes an edge between the graphs $graph(\widetilde{M}, \widetilde{\mathcal{M}})$ and $graph(\widetilde{\mathcal{M}}, \widetilde{\text{MOF}})$, which are meta-represented at the untyped metalevel. The *getMetaClass* function permits navigating edges of this kind.

- *container:* A model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, can be viewed as a tree $tree(\widetilde{M}, \widetilde{\mathcal{M}})$. When $\widetilde{M}$ is metarepresented at the MOF metalevel as $\widehat{M}$, such that $\widehat{M} = upObject(\widetilde{M})$, the function

$$container : [\![\textsc{Object}]\!]_{\text{MOF}} \times [\![\mathcal{M}^{\triangle}]\!]_{\text{MOF}} \times [\![\text{MOF}^{\triangle}]\!]_{\text{MOF}} \rightsquigarrow [\![\textsc{Object}]\!]_{\text{MOF}}$$

permits traversing $tree(\widetilde{M}, \widetilde{\mathcal{M}})$ at the untyped metalevel using a bottom-up strategy. The function *container* is defined for tuples of the form $(\widehat{o}, \widehat{M}, \widehat{\mathcal{M}})$, where:

$$\begin{aligned}
&\widehat{o} : \textsc{Object} \ \wedge\ \widehat{M} : \mathcal{M}^{\triangle} \ \wedge\ \widehat{\mathcal{M}} : \text{MOF}^{\triangle} \ \wedge \\
&downObject(\widehat{o}, \widetilde{\mathcal{M}}) \neq root(downModel(\widehat{M}, \widetilde{\mathcal{M}}), downModel(\widehat{\mathcal{M}}, \widetilde{\text{MOF}})).
\end{aligned}$$

The *container* function is defined by the equality

$$container(\widehat{o}, \widehat{M}, \widehat{\mathcal{M}}) = get_{MetaObject}(\widehat{o}, get_{String}(\widehat{prop}, "name", \widehat{\mathcal{M}}), \widehat{M}),$$

where:

$$\begin{aligned}
&\widehat{prop} : \textsc{Object} \;\wedge\; \widehat{prop} \in \widehat{\mathcal{M}} \;\wedge \\
&get_{Bool}(\widehat{prop}, \text{"}isComposite\text{"}, \widehat{\mathcal{M}}) = true \;\wedge \\
&get_{MetaObject}(\widehat{prop}, \text{"}class\text{"}, \widehat{\mathcal{M}}) \in \\
&\quad Bag\{getMetaClass(\widehat{o}, \widehat{\mathcal{M}})\} \;\cup\; superClasses(getMetaClass(\widehat{o}, \widehat{\mathcal{M}}), \widehat{\mathcal{M}});
\end{aligned}$$

and the function

$$superClasses : [\![\textsc{Object}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}^{\triangle}]\!]_{\mathrm{MOF}} \rightsquigarrow T_{\texttt{META-MODEL}, Bag\{MetaObject\}}$$

obtains the supertypes of the corresponding object type, and is defined as follows:

$$superClasses(\widehat{o}, \widehat{\mathcal{M}}) = get_{MetaObject}(\widehat{o}, \text{"}superClass\text{"}, \widehat{\mathcal{M}}) \cup \bigcup_{\widehat{o'} \in \left\{ \begin{array}{l} \widehat{o'} : \textsc{Object} \mid \\ \widehat{o'} \in get_{MetaObject}(\widehat{o}, \text{"}superClass\text{"}, \widehat{\mathcal{M}}) \end{array} \right\}} get_{MetaObject}(\widehat{o'}, \text{"}superClass\text{"}, \widehat{\mathcal{M}});$$

We also consider the function

$$containments : [\![\textsc{Object}]\!]_{\mathrm{MOF}} \times [\![\mathcal{M}^{\triangle}]\!]_{\mathrm{MOF}} \times [\![\mathrm{MOF}^{\triangle}]\!]_{\mathrm{MOF}} \rightsquigarrow T_{\texttt{META-MODEL}, Bag\{MetaObject\}}$$

that permits traversing the tree structure of a model definition following a top-down strategy. The *containments* function is defined for tuples of the form $(\widehat{o}, \widehat{M}, \widehat{\mathcal{M}})$, where: $\widehat{o}$: Object, $\widehat{M} : \mathcal{M}^{\triangle}$, $\widehat{\mathcal{M}}$: MOF$^{\triangle}$, and $downObject(\widehat{o})$ is *not* a leaf node in $tree(\widetilde{M}, \widetilde{\mathcal{M}})$. The *containments* function is defined as follows:

$$containments(\widehat{o}, \widehat{M}, \widehat{\mathcal{M}}) = \bigcup_{\widetilde{p} \in \left\{ \widetilde{p} : \textsc{Property} \mid \begin{array}{l} upObject(\widetilde{p}) \in \widehat{\mathcal{M}} \;\wedge \\ containment(\widetilde{p}, \widetilde{\mathcal{M}}) = true \end{array} \right\}} \begin{array}{l} asBag(get_{MetaObject}(\widehat{o}, \\ \quad get_{String}(upObject(\widetilde{p}), \text{"}name\text{"}, \widehat{\mathcal{M}}), \widehat{M})) \end{array}$$

where *asBag* corresponds to the OCL operation that converts a collection of elements into a bag of elements, and the *containment* function checks whether a property is defined as the opposite property of an *isComposite* property, as shown in Section 6.

The *containment* and *containments* functions are specified as equationally-defined operators in the `META-MODEL` theory as follows:

```
op _.‘container‘(_,_‘) :
    MetaObject ModelType{MetaObject} ModelType{MetaObject}
    ~> MetaObject .
op _.‘containments‘(_,_‘) :
    MetaObject ModelType{MetaObject} ModelType{MetaObject}
    ~> Bag{MetaObject} .
```

Taking into account the relational schema $\widetilde{\text{rsPerson}}$, which is shown at level M1 of the MOF framework in Fig. 2.2, such that $\widetilde{\text{rsPerson}}$: RDBMS, the container of the table $\widetilde{t}$, defined above, in $\widetilde{\text{rsPerson}}$ is a Schema instance, which can be obtained by reducing the following term:

$$\texttt{upObject(}\ \widetilde{t}\ \texttt{).container(}\ \widetilde{\text{rsPerson}}\ \texttt{,}\ \widetilde{\text{RDBMS}}\ \texttt{).}$$

- *isSet:* The function

$$isSet : [\![\textsc{Object}]\!]_{\mathrm{MOF}} \times [\![\textsc{String}]\!]_{\mathrm{MOF}} \rightsquigarrow [\![\textsc{Boolean}]\!]_{\mathrm{MOF}}$$

indicates whether a property is set in a specific Object instance or not. This function is defined for pairs of the form $(\widehat{o}, name)$, where $\widehat{o}$: Object, $name$: String, and $(property : name) \in getProperties(\widehat{o})$ or $(property : name = value) \in getProperties(\widehat{o})$, where $value$ is the value of the property. The *isSet* function is defined by means of the equality:

$$\widehat{o}\ .\ \mathit{isSet}(\mathit{PropName}) = \begin{cases} \mathit{true} & \mathit{if}\ (\mathit{property} : \mathit{PropName} = \mathit{value}) \in \\ & \qquad \mathit{getProperties}(\widehat{o}) \\ \mathit{false} & \mathit{if}\ (\mathit{property} : \mathit{PropName}) \in \mathit{getProperties}(\widehat{o}) \end{cases}$$

This function is specified in the `META-MODEL` theory as the equationally-defined operator

```
op _.‘isSet‘(_‘) : MetaObject String ~> Bool .
```

For the example, we can check whether the *column* property of the table `t` is set or not as follows:

$$\texttt{upObject(}\ \widetilde{t}\ \texttt{)}\ \ .\ \ \texttt{isSet("column")}.$$

When the property that is checked does not belong to the object type of the corresponding object, the *isSet* function remains undefined. For example, the property *type* does not belong to the TABLE object type, so that the resulting value is considered an undefined value, i.e.,

$$\texttt{upObject(}\ \widetilde{t}\ \texttt{)}\ \ .\ \ \texttt{isSet("type")}\ \ .\ \ \texttt{oclIsUndefined = true}.$$

- *unset:* The function

  $$\mathit{unset} : [\![\text{OBJECT}]\!]_{\text{MOF}} \times [\![\text{STRING}]\!]_{\text{MOF}} \rightsquigarrow [\![\text{OBJECT}]\!]_{\text{MOF}}$$

  deletes the value of a property, referred to by its name. This function is defined for pairs of the form $(\widehat{o}, \mathit{name})$, where $\widehat{o}$: OBJECT, *name* : STRING, and $(\mathit{property} : \mathit{name}) \in \mathit{getProperties}(\widehat{o})$ or $(\mathit{property} : \mathit{name} = \mathit{oldValue}) \in \mathit{getProperties}(\widehat{o})$, where *oldValue* is the value of the property. To define the function *unset*, we find two cases:

  – When the property to be unset is still *unset*, i.e.,

  $$(\mathit{property} : \mathit{name}) \in \mathit{getProperties}(\widehat{o}),$$

  the *unset* function is defined by the equation:

  $$\mathit{unset}(\widehat{o}, \mathit{name}) = \widehat{o}.$$

  – When the property to be unset is already *set*, i.e.,

  $$(\mathit{property} : \mathit{name} = \mathit{oldValue}) \in \mathit{getProperties}(\widehat{o}),$$

  the *unset* function is defined by the equation:

  $$\mathit{unset}(\widehat{o}, \mathit{name}) = \widehat{o}',$$

  where all the properties of $\widehat{o}$ are copied to $\widehat{o}'$ but $(\mathit{property} : \mathit{name}) \in \mathit{getProperties}(\widehat{o}')$ and $(\mathit{property} : \mathit{name} = \mathit{oldValue}) \notin \mathit{getProperties}(\widehat{o}')$.

  This function is specified in the `META-MODEL` theory as the equationally-defined operator

  ```
  op _.‘unset‘(_‘) : MetaObject String ~> MetaObject .
  ```

  In the example, the property `column` of the table `t` can be unset as follows:

  $$\texttt{upObject(}\ \widetilde{t}\ \texttt{)}\ \ .\ \ \texttt{unset("column")}$$

- *equals:* This operator indicates if two MOF OBJECT instances $\widehat{o1}$ and $\widehat{o2}$ are the same by comparing their identifiers. This operator is already provided as the OCL operator =. This operator is equationally-defined in the `MODEL{OBJ :: TH-OBJECT}` theory for `Object` terms. Since the sort `Object` of the `TH-THEORY` is mapped to the `MetaObject` sort of the `META-OBJECT` theory, by means of the `MetaObject` view, this operator is already defined for OBJECT instances, i.e., terms of sort `MetaObject`. This fact applies to the rest of OCL operators as well.

8.3 Summary

In this Section, we have formalized the MOF Reflection Facilities, which permit querying and manipulating any model definition by means of the operations of the Object object type. This object type introduces a metalevel, called untyped metalevel, where each model definition $\widetilde{M}$ that is typed with a given model type $\mathcal{M}$, at the metalevel, can be metarepresented independently of the corresponding metamodel realization $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$. Therefore, more generic functions can be defined at the untyped metalevel.

The operations of the Object object type are provided as equationally-defined operators in the `META-MODEL` theory of our framework, where each operation has a specific purpose:

- `getMetaObject:` permits traversing the graph structure of a model definition $\widetilde{M}$;
- `container/containments:` permit traversing the tree structure of a model definition $\widetilde{M}$;
- `getMetaClass:` provides introspection, so that the metadata that describes the object type of a given object can be queried; and
- `set:` provides structural reflection that permits manipulating a model definition $\widetilde{M}$.

In addition, the `OCL-COLLECTIONS{TH-OBJECT}{MetaObject}` and `MODEL{MetaObject}` theories are included in the `META-MODEL` theory, so that OCL operators can also be evaluated over metarepresented model definitions.

Part III

Applications

Chapter 9

Tools and Applications

In this Section, we provide a set of applications of the algebraic MOF framework that has been presented throughout Sections 5-8. Some of these applications are other modeling environments where models are managed as first-class citizens. These applications have provided the experimentation that is needed to validate our approach.

Taking into account the formalization of the MOF framework that has been presented in this work, we developed several tools for: (i) OCL constraint validation (MOMENT-OCL); (ii) model transformations by means of the QVT Relations language (MOMENT-QVT); and (iii) a preliminary version of a model management framework with traceability support (MOMENT). Some of these tools have been applied in other universities for academical purposes, and the MOMENT-QVT engine has also been applied in industry. The experience in the development of these tools and the experimentation with them has enabled the current specification of the algebraic MOF framework. These tools are based on an algebraic representation of metamodels that is introduced in [12]. The algebraic representation that has been presented in Section 6 is slightly different: it has been redesigned to take advantage of the object-oriented programming support of Maude, so that Maude-based formal verification facilities can be reused.

In subsequent sections, we describe: (i) the integration of the Algebraic MOF framework into informal MOF metamodeling environments, such as the Eclipse Modeling Framework (EMF); (ii) a brief description of the MOMENT-OCL tool; (iii) a brief description of the MOMENT-QVT tool, indicating its applications to other areas; (iv) the first experiments with the model management framework; and (v) an example of a graph rewriting system by using our algebraic MOF framework.

9.1 Interoperating Conventional and algebraic MOF frameworks

Informal MOF-based modeling frameworks provide a MOF metamodel implementation $\mathrm{MOF}_{\mathrm{MOF}}$ that may include the metamodel of OCL-like languages. Some of these modeling frameworks have an informal implementation of the reflection mechanism $\widetilde{\mathcal{M}_{\mathrm{MOF}}} \mapsto \mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}_{\mathrm{MOF}}})$, which realizes a metamodel as a program in a conventional language. The resulting metamodel application usually consists of an editor for models $\widetilde{M_{\mathrm{MOF}}} : \mathcal{M}_{\mathrm{MOF}}$, having facilities for model serialization to XMI, repository functionality, graphical representation, informal text generation or informal model transformations.

To formalize conventional modeling environments, we provide a generic bidirectional function σ that merely performs a syntactic representation change for metamodel specification definitions $(\widetilde{\mathcal{M}_{\mathrm{MOF}}}, \widetilde{\mathcal{C}_{\mathrm{MOF}}}) \mapsto (\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, where $\widetilde{\mathcal{M}} : (\mathrm{MOF}, \mathcal{C}_{\mathrm{MOF}})$ and $\widetilde{\mathcal{C}} : (\mathrm{OCL}, \mathcal{C}_{\mathrm{OCL}})$ in our algebraic framework. Given a metamodel definition $\widetilde{\mathcal{M}_{\mathrm{MOF}}}$ in an informal MOF framework and a model definition $\widetilde{M_{\mathrm{MOF}}}$ that conforms to $\widetilde{\mathcal{M}_{\mathrm{MOF}}}$, the function σ is defined by the mapping $\widetilde{M_{\mathrm{MOF}}} \mapsto \widetilde{M}$, where $\widetilde{M} : \mathcal{M}$ and $\mathcal{M}$ represents the model type that corresponds to $\widetilde{\mathcal{M}_{\mathrm{MOF}}}$. The function σ is extended to metamodel specification definitions of the form $(\widetilde{\mathcal{M}_{\mathrm{MOF}}}, \widetilde{\mathcal{C}_{\mathrm{MOF}}})$ as follows:

$$\sigma(\widetilde{\mathcal{M}_{\mathrm{MOF}}}, \widetilde{\mathcal{C}_{\mathrm{MOF}}}) = (\sigma(\widetilde{\mathcal{M}_{\mathrm{MOF}}}), \sigma(\widetilde{\mathcal{C}_{\mathrm{MOF}}})).$$

The function σ is also easily defined for sets of OCL constraint definitions $\widetilde{\mathcal{C}_{\mathrm{MOF}}}$ as follows:

$$\sigma(\widetilde{\mathcal{C}}_{\mathrm{MOF}}) = \bigcup_{\widetilde{c}_{\mathrm{MOF}} \in \widetilde{\mathcal{C}}_{\mathrm{MOF}}} \sigma(\widetilde{c}_{\mathrm{MOF}}),$$

where $\widetilde{c}_{\mathrm{MOF}} : \mathrm{OCL}_{\mathrm{MOF}}$, $\sigma(\widetilde{c}_{\mathrm{MOF}}) : (\mathrm{OCL}, \mathcal{C}_{\mathrm{OCL}})$, and $\mathrm{OCL}_{\mathrm{MOF}}$ is the definition of the OCL metamodel in the informal MOF framework. Therefore, $\widetilde{c}_{\mathrm{MOF}}$ can be translated by means of the function σ as any other model definition. Once a metamodel specification definition $(\widetilde{\mathcal{M}_{\mathrm{MOF}}}, \widetilde{\mathcal{C}_{MOF}})$ is formally represented as data $\sigma(\widetilde{\mathcal{M}_{MOF}}, \widetilde{\mathcal{C}_{\mathrm{MOF}}})$, its algebraic semantics is provided by the MEL theory *reflect*$(\sigma(\widetilde{\mathcal{M}_{\mathrm{MOF}}}, \widetilde{\mathcal{C}_{\mathrm{MOF}}}))$.

The function σ constitutes a bijection that can also map a formal data representation of a metamodel or a model into their corresponding informal representation. This bidirectional bridge establishes a separation layer between informal modeling frameworks and our MOF formal framework. On the one hand, σ and σ^{-1} provide a framework where theoreticians can work in the formalization of model-based techniques without the requirement of knowing the ever-changing technology that is based on standards like MOF, UML, OCL, XMI or XML. On the other hand, σ^{-1} and σ allow considering the mentioned standards as well-defined interfaces to apply formal techniques in a transparent way.

9.1.1 Interoperating the EMF and our MOF Algebraic Framework

The Eclipse Modeling Framework (EMF) provides a close implementation to the MOF metamodel, which is called the Ecore metamodel. The initial purpose of EMF was to unify XML, Java and UML technologies. The EMF provides an informal implementation of the reflection mechanism that permits obtaining the metamodel realization $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}_{\mathrm{ECORE}}})$ from an Ecore metamodel definition $\widetilde{\mathcal{M}_{\mathrm{ECORE}}}$. The EMF metamodel realization $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}_{\mathrm{ECORE}}})$ also offers a tree-like editor to define models. The EMF implements the MOF reflection facilities in the EObject and EFactory classes and provides XMI 2.0 serialization support. Furthermore, the *Eclipse Modeling Project* provides OCL support for Ecore metamodels and the project *Graphical Modeling Framework* permits attaching a graphical representation to a specific Ecore metamodel.

The mapping between the EMF and our Maude-based framework for MOF is provided by the bijection σ. The σ function maps a model $\widetilde{M_{EMF}}$ that has been defined in the EMF to a term of sort *ModelType*{*MetaObject*}, i.e., to the metarepresentation $\widehat{M}$ of a model definition $\widetilde{M} : \mathcal{M}$, where $\mathcal{M}$ is the corresponding model type that is defined as an Ecore model in the EMF. σ is implemented as a set of code templates that are applied by a code template engine to obtain the term $\widehat{M} : \mathcal{M}^{\triangle}$. σ is a generic mapping that can be applied to any Ecore metamodel:

- To obtain the data representation of a metamodel $\widetilde{\mathcal{M}} : \mathrm{ECORE}$.
- To obtain the data representation of an OCL constraint $\widetilde{c} : \mathrm{OCL}$.
- To obtain the data representation of a model that conforms to another EMF metamodel $\widetilde{M} : \mathcal{M}$.

Note that we use the ECORE metamodel as the implementation of the EMOF meta-metamodel.

Our algebraic MOF framework as an Eclipse plugin

The σ function is implemented in an Eclipse plugin, which permits representing an EMF model definition $\widetilde{M_{EMF}}$ as a term of sort *ModelType*{*MetaObject*}. This plugin is available in [92]. Once installed, the plugin adds a menu, called *AlgebraicMOF*. When we choose a XMI file that contains an EMF model definition, this menu provides several functionalities:

- *ToMetaObjectConfiguration*, (3) in Fig. 9.1: applies the σ function to an EMF model definition $\widetilde{M}$, such that $\widetilde{M_{EMF}} \mathrel{\widetilde{:}} \widetilde{\mathcal{M}_{EMF}}$, obtaining its metarepresentation $\widehat{M} : \mathcal{M}^{\triangle}$. In this case, $\widetilde{\mathcal{M}_{EMF}}$ corresponds to the metamodel definition in the EMF framework, and $\widetilde{\mathcal{M}}$ to the metamodel definition in the algebraic MOF framework. Note that in the EMF, the conformance relation $\widetilde{M_{EMF}} \mathrel{\widetilde{:}} \widetilde{\mathcal{M}_{EMF}}$ can only be characterized as data.
- *FromMetaObjectConfiguration*, (2) in Fig. 9.1: applies the σ^{-1} function to a model definition $\widetilde{M} : \mathcal{M}$ that is metarepresented as $\widehat{M} : \mathcal{M}^{\triangle}$ in the algebraic MOF framework. This functionality is enabled for files with extension *.maude*. A file of this kind must contain a first line with the URI that identifies the corresponding metamodel definition, and the term representing $\widehat{M}$. The uri has to be given using a specific format:

$$\texttt{***\$}\ \boxed{\mathit{nsPrefix}}\ \texttt{-}\ \texttt{"}\boxed{\mathit{nsUri}}\texttt{"}$$

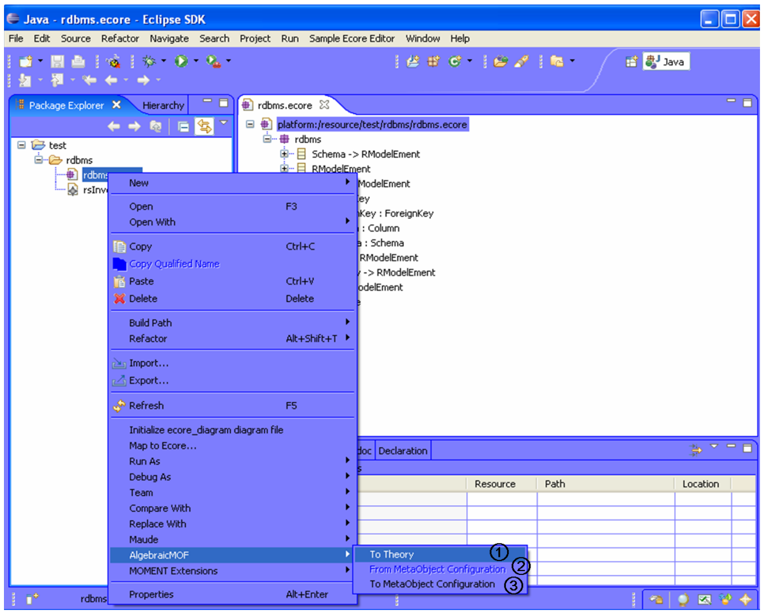


Figure 9.1: The algebraic MOF framework into the Eclipse platform.

where ***nsPrefix*** represents the value of the *nsPrefix* attribute of the root PACKAGE instance of the metamodel definition $\widetilde{\mathcal{M}_{EMF}}$, and ***nsUri*** represents the value of its *nsUri* attribute. For example, to parse a model definition $\widetilde{M}$: ECORE that is metarepresented as $\widehat{M}$: ECORE$^{\triangle}$, we must add the following line:

```
***$ ecore - "http://www.eclipse.org/emf/2002/Ecore"
```

- *ToTheory*, (1) in Fig. 9.1: applies the $\mathit{reflect}_{\mathrm{MOF}}$ function to a metamodel definition $\widetilde{\mathcal{M}_{EMF}}$. In fact, the obtained result corresponds to the theory $\mathit{reflect}_{\mathrm{MOF}}(\mathit{downModel}(\sigma(\widetilde{\mathcal{M}_{EMF}}), \widetilde{\textsc{Ecore}}))$, which is represented as a Maude module.

To apply the σ and σ^{-1} functions to a model definition $\widetilde{M_{EMF}}$, such that $\widetilde{M_{EMF}} \;\tilde{:}\; \widetilde{\mathcal{M}_{EMF}}$, the corresponding metamodel definition $\widetilde{\mathcal{M}_{EMF}}$ must be available as a plugin, see [98] for further details on using EMF.

Taking into account that we use the ECORE metamodel as the implementation of the MOF metamodel, recall the metamodel definition $\widetilde{\mathrm{RDBMS}}$, such that $\widetilde{\mathrm{RDBMS}}$: ECORE, and the model definition $\widetilde{\textsc{rsPerson}}$, such that $\widetilde{\textsc{rsPerson}}$: RDBMS, that were defined in Fig. 2.2.

On the one hand, to obtain the $\widetilde{\mathrm{RDBMS}}$ metamodel definition, we apply the *ToMetaObjectConfiguration* method to the file that contains its definition in EMF. This provides the metarepresentation $\widehat{\mathrm{RDBMS}}$: ECORE$^{\triangle}$ of the model definition $\widetilde{\mathrm{RDBMS}}$: ECORE, which is shown in Appendix B. If we apply the *FromMetaObjectConfiguration* method to the file that is obtained by means of *ToMetaObjectConfiguration*, we recover the original metamodel definition in EMF. Applying the method *ToTheory* obtains the theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{RDBMS}})$, so that model definitions $\widetilde{M}$: RDBMS can be defined. This theory is provided in Appendix C. The Maude module that represents the MEL theory $\mathit{reflect}(\widetilde{\textsc{Ecore}}, \varnothing)$ is provided in Appendix A.

On the other hand, when we apply the *ToMetaObjectConfiguration* method to the model definition $\widetilde{\textsc{rsPerson}}$, we obtain the metarepresentation $\widehat{\textsc{rsPerson}}$: RDBMS$^{\triangle}$, which is shown in Appendix E. If we apply the *FromMetaObjectConfiguration* to the resulting file, we obtain the original model definition in EMF.

Once the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{RDBMS}})$ theory is obtained, we can apply the `downModel` operator as follows:

```
red downModel( model, mm(nullObject#rdbms), nullObject#rdbms ) .
```

where `model` is a constant that represents the metarepresented model definition $\widehat{\textsc{rsPerson}}$ that is obtained by means of *ToMetaObjectConfiguration*.

This plugin depends on another plugin, called *Maude Development Tools*, that integrates Maude into the Eclipse platform. The *Maude Development Tools* plugin offers a Java library that permits interacting with Maude from Java code, and a set of editing facilities to develop Maude programs within the Eclipse platform. This plugin has been developed as part of the MOMENT Project and is available at [92].

Pending work

In this work, we have provided a detailed definition of the *reflect* function that permits projecting a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ as a theory, enabling the definition $\widetilde{M}$ of models that both conform to $\mathcal{M}$, $\widetilde{M} : \mathcal{M}$, and satisfy the OCL constraints $\widetilde{\mathcal{C}}$, $\widetilde{M} \models \mathcal{C}$. In the current implementation, we provide an specification of the $\mathit{reflect}_{\mathrm{MOF}}$ function, which is used in the *reflect* function. The $\mathit{reflect}_{\mathrm{MOF}}$ function is the function used in the *ToTheory* method.

Despite the algebraic specification of the generic semantics of OCL operators and the mathematical definition of the *reflect* function that is provided in Section 7, OCL constraint definitions cannot be given using the concrete syntax of the OCL language in our current implementation. This feature is provided by means of the *reflect* function. To define the *reflect* function, we have taken advantage of our previous experience with a prototype for OCL constraint validation, called *MOMENT-OCL* [11]. *MOMENT-OCL* uses the OCL support of the Kent Modeling Framework [99] to parse OCL expressions that are given in textual format, and traverses the abstract syntax tree of an OCL expression, generating a term that represents the OCL expression in Maude, in a similar way as we do in this work. In future work, we will define a new version of the *MOMENT-OCL* tool, using the Algebraic MOF framework and specifying the *reflect* function that we have defined in this work. As front-end, we will use the OCL support of the *Model Development Tools* project [100], which defines the abstract syntax of the OCL language as an EMF metamodel, and provides a parser for OCL expressions.

Another choice that we will also take into account to provide support for the concrete syntax of the OCL language, consists in defining the semantics of the OCL language as indicated in the *Rewriting Logic Semantics* project [101, 102]. The first choice permits defining the semantics of the OCL language following a model-based approach, where the concrete syntax of the OCL language is provided by the EMF-based front-end, OCL constraint definitions are provided as model definitions in EMF, and the formal semantics of the OCL language is provided in MEL and specified in Maude. However, if we want to reuse the specification of the OCL language for other purposes, like formal analysis or model transformations, we depend on the front-end implementation. The second choice permits defining both the syntax and semantics of the OCL language in Rewriting Logic directly, and thereby has a direct implementation in Maude. In the latter case, model-based support is not given directly but the concrete syntax of the OCL language is available as MEL signatures, so that OCL support can be easily reused for other purposes.

Considering other Modeling Frameworks

Not just the EMF can be mapped to our algebraic MOF framework by means of σ: other MOF-like modeling frameworks based on similar concepts can also be mapped to our algebraic MOF framework. Their main differences are usually syntactic. Two approaches can be followed to reuse our specification in other modeling frameworks:

- Integrating Maude into the corresponding framework and redeveloping σ to map concepts from the corresponding informal meta-metamodel to the formalization of the Ecore meta-metamodel.
- Defining a foreign meta-metamodel, MOF', in EMF and defining a bidirectional model transformation between MOF' and Ecore meta-metamodels. A model transformation can be viewed as an equationally-defined function

  $$f : \mathit{ConsistentConfiguration}\{\text{MOF'}\} \rightarrow \mathit{ConsistentConfiguration}\{\textsc{Ecore}\}$$

  that changes the syntactical representation of a metamodel specification by means of the mapping $f : \widetilde{\mathcal{M}_{\text{MOF'}}} \mapsto \widetilde{\mathcal{M}_{\textsc{Ecore}}}$, where $\widetilde{\mathcal{M}_{\text{MOF'}}}$: MOF' and $\widetilde{\mathcal{M}_{\textsc{Ecore}}}$: ECORE. Some experiments in mapping meta-metamodels have been already reported in [103], where the authors map the meta-metamodel of the *Domain-Specific Languages tools* framework and the Ecore meta-metamodel of the EMF. Therefore, Ecore is used as a pivot meta-metamodel and σ can be reused as is.

Both approaches should take into account the transformation of constraints that are specified in different constraint definition languages. For example, LINQ in the DSL tools and OCL in the EMF. Nevertheless,

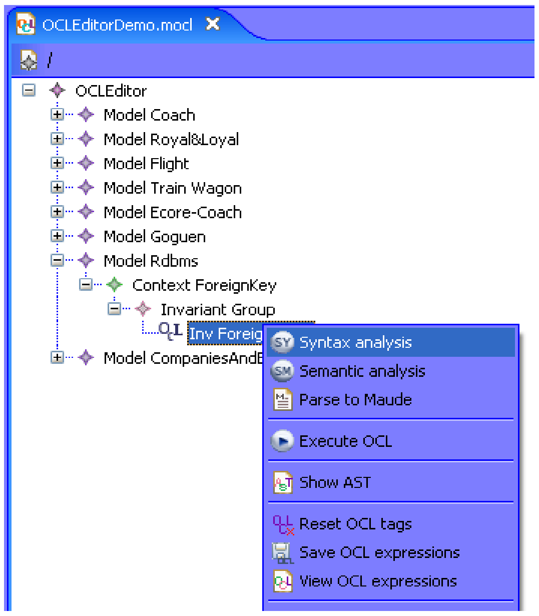


Figure 9.2: Tree editor of the MOMENT-OCL tool.

metamodel-based support is not always available for languages of this kind, which are usually grammar-based. In these cases, alternative ad-hoc solutions to model transformations are needed to traverse the corresponding abstract syntax trees of constraint expressions.

9.2 MOMENT-OCL

MOMENT-OCL [11] is a tool for OCL constraint validation and OCL query evaluation that is integrated into the EMF. An EMF model definition $\widetilde{M}_{EMF}$ conforms to a metamodel definition $\widetilde{\mathcal{M}}_{EMF}$ that is defined as an Ecore model definition, where Ecore is the meta-metamodel of the EMF. MOMENT-OCL follows the philosophy that has been presented in Section 7 to represent OCL expressions as terms in an algebraic theory. This theory is generated from a pair $(\widetilde{\mathcal{M}}_{EMF}, \textit{ocl-expression})$, where $\widetilde{\mathcal{M}}_{EMF}$ is a metamodel definition in the EMF, and *ocl-expression* is either an OCL constraint or an OCL query that is given in textual form. MOMENT-OCL uses the OCL library of the Kent Modeling Framework [99] as front-end to parse OCL expressions, and the EMF as the front-end for the metamodeling framework. MOMENT-OCL provides the following functionality:

- *Definition of OCL expressions,* allowing the definition of OCL invariants and OCL queries. OCL expressions are defined for a specific metamodel definition in EMF. Fig. 9.2 shows the tree editor that constitutes the MOMENT-OCL tool interface. In the editor, we can add *model* nodes, representing specific metamodel definitions. A *model* node has properties, by means of which the metamodel definition and the model definition to be queried are referred to. These properties are shown in the common EMF Properties view. Within a *model* node, the user can define *context* nodes, representing the context of OCL expressions. In a *context* node, the user can add *OCL invariants* or *OCL queries*. All the OCL expressions that are defined within a *context* node have the same contextual type, indicated in a property of the corresponding *context* node.

- *Syntactic and semantic analysis,* indicating whether an OCL expression is well-formed or not, and whether an OCL constraint is meaningful for a specific metamodel definition or not. These tasks are performed by the OCL library of the Kent Modeling Framework.

- *Generation of the algebraic representation of OCL expressions,* providing the Maude code that represents the operators and terms that are generated for a given OCL expression. Note that the generated code is not exactly the same that has been presented in Section 7, due to the aforementioned representational differences.

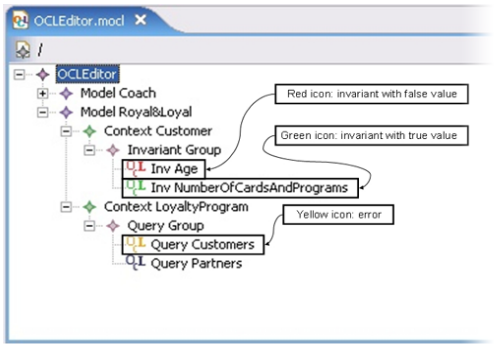
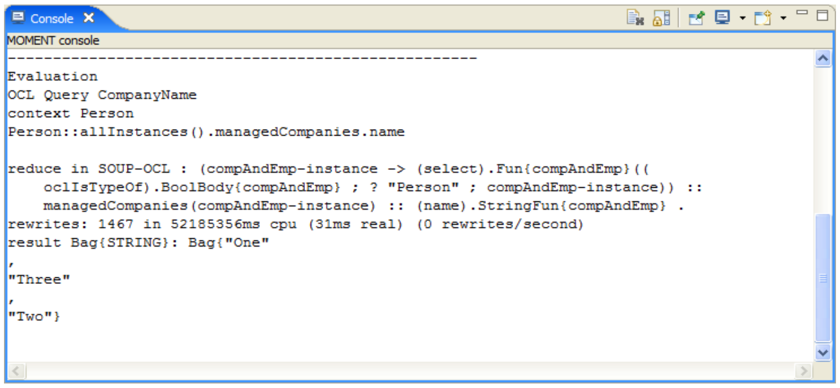


Figure 9.3: Evaluation of OCL expressions in the MOMENT-OCL tool.



```
------------------------------------------------------
Evaluation
OCL Query CompanyName
context Person
Person::allInstances().managedCompanies.name

reduce in SOUP-OCL : (compAndEmp-instance -> (select).Fun{compAndEmp}((
    oclIsTypeOf).BoolBody{compAndEmp} ; ? "Person" ; compAndEmp-instance)) ::
    managedCompanies(compAndEmp-instance) :: (name).StringFun{compAndEmp} .
rewrites: 1467 in 52185356ms cpu (31ms real) (0 rewrites/second)
result Bag{STRING}: Bag{"One"
,
"Three"
,
"Two"}
```

Figure 9.4: Console view of the MOMENT-OCL tool.

- *Execution of OCL invariants and OCL queries.* In the tree view, when an OCL invariant is evaluated over a specific model definition $\widetilde{M}$, referred to in the corresponding *model* node, the icon of the corresponding *OCL invariant* node in the tree view changes its color indicating the result, as shown in Fig. 9.3: *red* if the invariant has failed, *green* if the invariant has succeeded, and *yellow* if there has been an error during the evaluation of the OCL constraint. As explained in Section 7, an OCL expression is represented by a term, whose equational simplification, by using the equations that are generated for the pair $(\widetilde{\mathcal{M}}_{EMF}$, *ocl-expression*$)$ results in a canonical form that represents the resulting value of the OCL expression. The result that is obtained by the evaluation of an OCL invariant or an OCL query, i.e., the resulting term, is shown in the *console view* of the tool, shown in Fig. 9.4. Despite the algebraic representation of the resulting term, it is shown in the concrete syntax format of the OCL language, thanks to the mixfix notation that is supported by Maude.
- *Persistence of OCL expressions,* in textual format.

Fig. 9.5 shows the components of the MOMENT-OCL prototype that permit the execution of algebraic OCL expressions over EMF model definitions:

- The OCL Projector component is the module that projects the OCL expression to Maude code. It makes use of the Kent OCL library [99] to validate the syntax and the semantics of the expression. The process of compilation from OCL to Maude follows the typical structure of a language processor. The process is divided in two phases: an initial analysis phase and a second synthesis phase.

  In the first phase, we have reused the OCL support of the Kent Modeling Framework (KMF), which provides lexical, syntactic and semantic analysis of OCL expressions over an EMF model definition. KMF analyzes an OCL expression, taking into account the semantics of the model definition, and produces an Abstract Syntax Tree (AST) to represent the data that is needed in the synthesis phase.

  In the second phase, once an OCL expression has been analyzed by KMF correctly, the AST is traversed and Maude code for body expressions, queries and invariants, is produced in order to evaluate OCL expressions over EMF model definitions. In MOMENT-OCL, the functions *getExpTheory* and *getExpTerm* that are defined in Section 7 are likewise implemented in Java. However, instead of traversing

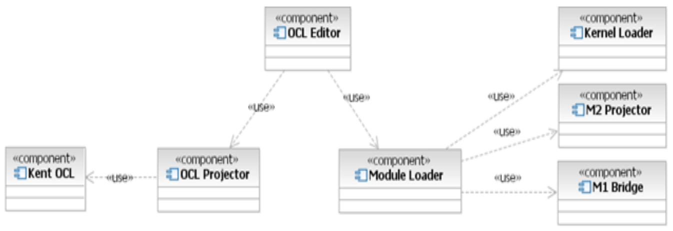


Figure 9.5: Overview of the architecture of the MOMENT-OCL tool.

a model definition $\widetilde{c}$: OCL that represents the corresponding OCL expression, MOMENT-OCL traverses the AST of the OCL expression, provided by KMF. Another difference with the specification of the OCL language that is given in Section 7 is that *MOMENT-OCL* does not support undefined values in OCL expressions.

- The Module Loader component obtains the algebraic specification from an EMF metamodel. This algebraic specification is extended with the Maude code obtained from the compilation of OCL expressions by means of the OCL Projector component. The Module Loader uses three other components: the M2 Projector, which obtains the algebraic representation of a metamodel definition; the M1 Bridge, which projects a model definition $\widetilde{M}$ as a term of the corresponding algebraic theory; and the Kernel Loader, which loads the corresponding Maude modules, providing the formal environment where OCL expressions can be evaluated over the model definition $\widetilde{M}$.
- The OCL Editor permits the definition of OCL queries and invariants over EMF model definitions, and provides syntactic and semantic analysis of the expressions by reusing this functionality from the KMF, as indicated above.

9.3 MOMENT-QVT

In MDA, model transformations have become a relevant issue by means of the standard Query/Views/Transformations (QVT) [23]. As indicated in [104], since a software artifact can be viewed as a model definition, model transformation is the basic mechanism that permits the manipulation of software artifacts.

In this section, we focus on the MOMENT-QVT tool, a model transformation engine that provides partial support for the QVT Relations language. This tool is based on an algebraic operator, the ModelGen operator, whose axioms are generated from a model transformation defined by using the QVT Relations language. Some first experiments with model transformations were provided in [3], and the model transformation engine was proposed in [5, 6]. Since the ModelGen operator is algebraically specified in Maude, this term rewriting system is used as the underlying runtime environment for model transformations in MOMENT. This fact provides an efficient environment to execute the ModelGen operator.

9.3.1 The QVT Relations Language and the *ModelGen* Operator

In the QVT Relations language, a model transformation is defined among several metamodels, which are called the domains of the transformation. A QVT transformation is constituted by QVT relations, which become declarative transformation rules. A QVT relation specifies a relationship that must hold between the model objects of different candidate model definitions. The direction of the transformation is defined when it is invoked by choosing a specific domain as target. If the target domain is defined in the QVT transformation as *enforce*, a transformation is performed by creating the corresponding elements in the target model definition. If the target domain is defined as *checkonly*, just a checking is performed, without creating any new element in the target model definition. Both kinds of transformations are used in our approach.

A relation can also be constrained by two sets of predicates, a *when* clause and a *where* clause. The *when* clause specifies the conditions under which the relation needs to hold. The *where* clause specifies the condition that must be satisfied by all model objects participating in the relation.

A transformation contains two kinds of relations: top-level (marked with the *top* keyword) and non-top-level. The execution of a transformation requires that all its top-level relations hold, whereas non-top-level relations are required to hold only when they are invoked directly or transitively from the *where* clause of another relation.

As example, we have taken the *UmlToRdbms* transformation that is presented in the MOF QVT final specification[1]. The top relation below specifies the transformation of a *Class* into a *Table*. By means of the *where* clause, the relation *ClassToTable* needs to hold only when the *PackageToSchema* relation holds between the package containing the class and the schema containing the table. By means of the *when* clause, the *ClassToTable* relation holds, the relation *AttributeToColumn* must also hold.

```
top relation ClassToTable {
    className: String;
    checkonly domain ecoreDomain c: EClass {
        ePackage = p:EPackage {},
        name=className
    };
    enforce domain rdbmsDomain t: Table {
        schema = s:Schema {},
        name = className,
        column = cl:Column {
            name = className + '_tid',
            type = 'NUMBER'
        },
        key = k:Key {
            name = className + '_pk',
            column=cl
        }
    };
    when {
        PackageToSchema(p, s);
    }
    where {
        AttributeToColumn(c, t, className);
    }
}
```

In MOMENT-QVT, a model transformation can be applied to several source model definitions, which may or may not conform to the same metamodel. When the transformation is invoked, it generates one target model definition and a set of traceability model definitions. A traceability model definition contains a set of traces that relate the elements of the source model definition to the elements of the target model definition, indicating which transformation rule has been applied to each source element. A QVT Relations enforced transformation is executed by means of the *ModelGen* operator as follows:

$$< output\_model,\ trac_1, ...,\ trac_n >= ModelGen(transformation,\ input\_model_1, ...,\ input\_model_n)$$

where $transformation$ is the name of the QVT transformation; $input\_model_1, ...,\ input\_model_n$ are the input model definitions, which may conform to different metamodels; $output\_model$ is the generated model definition; and $trac_1, ...,\ trac_n$ are the trace model definitions that are generated for each one of the corresponding input model definitions.

9.3.2 Overview of a model transformation in MOMENT-QVT

In the QVT Relations language, a model transformation is defined among several metamodels, which are called the domains of the transformation. A QVT transformation is constituted by QVT relations, which become declarative transformation rules. A QVT relation specifies a relationship that must hold between the model objects of different candidate model definitions. The direction of the transformation is defined when it is invoked by choosing a specific domain as target. If the target domain is defined in the QVT transformation as *enforceable*, a transformation is performed. If the target domain is defined as *checkonly*, just a checking is performed.

In MOMENT, a QVT transformation is defined by means of the *ModelGen* operator. QVT relations are defined by means of the *ModelGenRule* operator , which is used by the former operator. The generation of the axioms for the *ModelGen* operator and *ModelGenRule* operator are provided in [9]. A model transformation can be applied to several source model definitions, which may or may not conform to the same metamodel. It generates one target model definition and a set of traceability model definitions. A traceability model

[1]We used a version of this transformation in which we consider Ecore as an implementation of the UML Class Diagram metamodel.

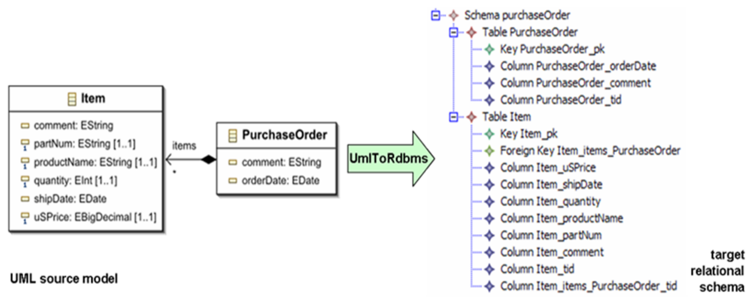


Figure 9.6: Example of Model Transformation.

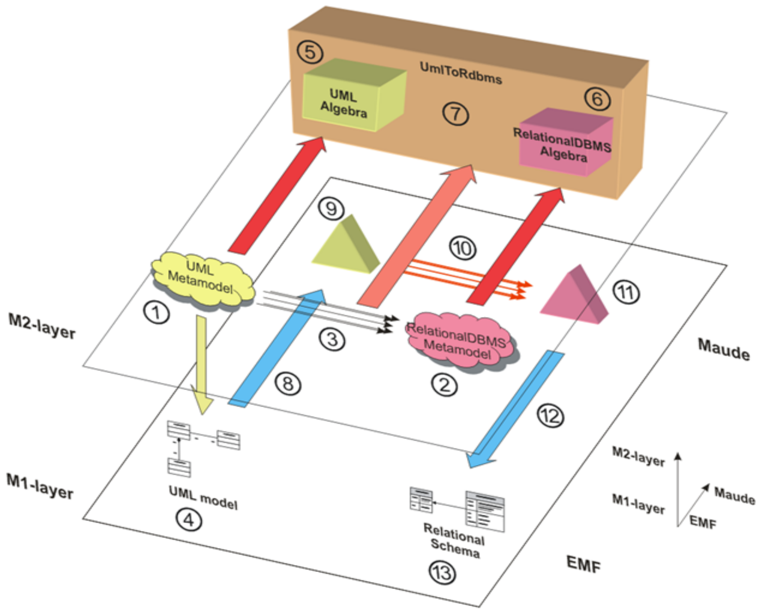


Figure 9.7: Example of Model Transformation.

definition contains a set of traces that relate the elements of the source model definition to the elements of the target model definition, indicating which transformation rule has been applied to each source element.

We have chosen the *UmlToRdbms* transformation that is presented in the QVT final adopted specification [23] as an example to illustrate the use of the *ModelGen* operator in the MOMENT Framework. The ECORE metamodel [98] has been used as implementation of the UML metamodel. The RDMBS metamodel of the QVT proposal has been specified as an EMF metamodel. Using both metamodels, the *UmlToRdbms* transformation is applied to the source ECORE model definition in Fig. 9.6 to obtain the target relational schema, which is shown in the figure by using the default EMF graphical modeler.

In this section, we present how MOMENT executes the *ModelGen* operator, transforming the UML model definition of the example in Section 2 into a relational schema. Fig. 3 shows the two MOF layers involved in a model transformation: the M2-layer, where the metamodels are defined; and the M1-layer, where the model transformation and the model definitions are defined and manipulated. The front part of the figure represents the front-end of the MOMENT framework, i.e., EMF and all the plugins that are built on it. The back part of the figure represents the formal back-end of the MOMENT Framework, where Maude is used. Traceability support has not been taken into account in the figure.

Fig. 9.7 represents the transformation of the UML model definition by using the ModelGen operator. The steps that are automatically performed by the MOMENT Framework when the ModelGen operator is applied to the source UML model definition are the following:

- *(1) and (2):* We specify both UML and RelationalDMBS metamodels at the M2-layer by means of the EMF or graphical editors based on this modelling framework. For instance, we can also consider

XML schemas and Rational Rose model definitions as metamodels.

- *(3):* The QVT transformation is defined as a model definition at the M1-layer, but it relates the constructs of the source to the constructs of the target metamodels. The transformation has to be defined as a model definition that conforms the QVT Relations metamodel by means of a graphical interface or as a program using the Relations language. The transformation model definition can either be defined by the user or be automatically produced by another transformation.
- *(4):* We define a UML model definition using a UML graphical editor based on EMF.
- *(5) and (6):* Both UML and RDBMS metamodels, respectively, are projected as algebraic specifications by means of the interoperability bridges that have been implemented in the MOMENT framework. This bridge corresponds to the Java implementation of the $\mathit{reflect}_{\mathrm{MOF}}$ function, presented in Section 6. However, the obtained syntactic representation of metamodels is different as indicated above.
- *(7):* The model that defines the QVT transformation is projected into the Maude code as the *UmlToRdbms* module, which contains the specification of the *ModelGen* and *ModelGenRule* operators.
- *(8):* The source UML model definition, which is defined in step *(4)* at the M1-layer, is projected as a term of the UML theory *(9)*.
- *(10):* Maude applies the *ModelGen* operator through its equational deduction mechanism, obtaining a term of the RDBMS theory *(11)*. Thus, Maude constitutes the runtime engine for the MOMENT transformation mechanism.
- *(12):* This is the last step of the model transformation process. It parses the term *(11)*, defining an EMF model definition *(13)* in the M1-layer, which conforms to the target metamodel defined at the M2-layer.

In the model transformation process, the user only interacts with the MOMENT framework when defining the source and target metamodels (steps *(1)* and *(2)*), the QVT transformation between both metamodels (step *(3)*) and the source model definition (step *(4)*). The other steps are automatically carried out by the framework. The output model definition can also be manually manipulated from a graphical editor.

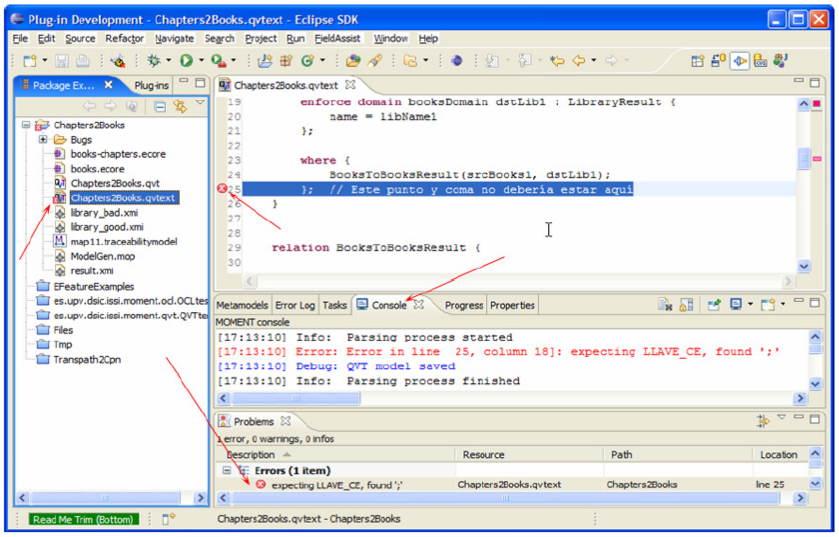
9.3.3 MOMENT-QVT

MOMENT-QVT implements the metamodel definition $\widetilde{\mathrm{QVT}}$, given in the QVT standard, and provides an editor for the QVT Relations language, which permits defining model transformations between EMF metamodels. Fig. 9.8 shows the editor of the MOMENT-QVT tool which provides: syntax coloring, editing facilities and parsing facilities. For example, when a model transformation definition is not well-defined, the editor indicates which line contains the error.

Once the model transformation is defined by using the concrete syntax of the QVT Relations language, we have to parse it as shown in Fig. 9.9, generating a QVT model definition $\widetilde{M} : \mathrm{QVT}$. QVT is the model type that corresponds to the metamodel definition $\widetilde{\mathrm{QVT}}$.

After obtaining the model definition $\widetilde{M} : \mathrm{QVT}$ that corresponds to the user-defined model transformation, the user can invoke the model transformation by using the invocation wizard that is shown in Fig. 9.10. To invoke a model transformation, the user has to choose the *ModelGen* operator in the *Operator name* panel. After this, the user has to provide the file that contains the model definition $\widetilde{M} : \mathrm{QVT}$. Depending on the definition of the model transformation, the user has to provide the input parameters and the output parameters. For the example, the input parameter is the source Ecore model definition that represents the source UML model definition, and the output parameters are the file that will contain the resulting RDBMS model definition and the traceability model definition.

MOMENT-QVT provides support for traceability, in the sense that a traceability model definition, which records what objects of the target model definition have been generated from objects of the source model definition, is generated in an automated way during a model transformation. Fig. 9.11 presents the traceability editor of the MOMENT framework. This editor shows the traceability model definition that is generated by the *UmlToRdbms* transformation. The traceability editor is constituted by three main frames, the left frame shows an input model definition of the transformation, the right frame shows the output generated model definition, and the frame in the middle shows the traces that relate elements of the input model definition to elements of the target model definition. Traces also provide information about the transformation rule (or relation) that has been applied to source objects to generate the corresponding target objects.



```
        enforce domain booksDomain dstLib1 : LibraryResult {
            name = libName1
        };

        where {
            BooksToBooksResult(srcBooks1, dstLib1);
        };  // Este punto y coma no debería estar aquí
    }


    relation BooksToBooksResult {
```

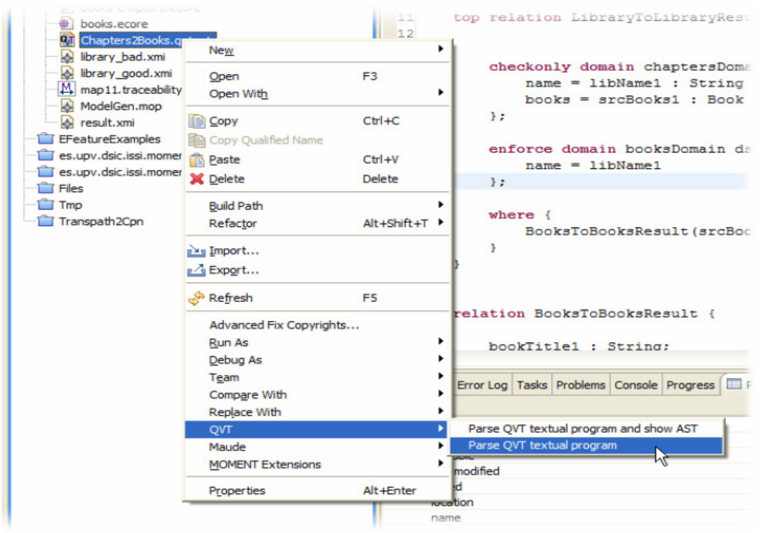
```
[17:13:10] Info:  Parsing process started
[17:13:10] Error: Error in line  25, column 18]: expecting LLAVE_CE, found ';'
[17:13:10] Debug: QVT model saved
[17:13:10] Info:  Parsing process finished
```

1 error, 0 warnings, 0 infos

Description	Resource	Path	Location
Errors (1 item)			
expecting LLAVE_CE, found ';'	Chapters2Books.qvtext	Chapters2Books	line 25

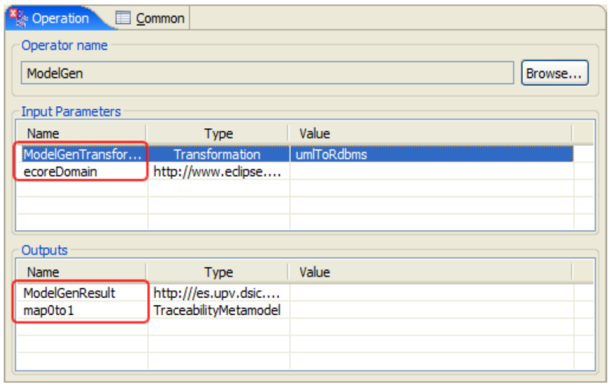
Chapters2Books.qvtext - Chapters2Books

Figure 9.8: Defining a model transformation.



```
    top relation LibraryToLibraryRes

        checkonly domain chaptersDoma
            name = libName1 : String
            books = srcBooks1 : Book
        };

        enforce domain booksDomain ds
            name = libName1
        };

        where {
            BooksToBooksResult(srcBoo
        }
    }


    relation BooksToBooksResult {

        bookTitle1 : String;
```

Figure 9.9: Parsing a model transformation definition.



Operator name: ModelGen

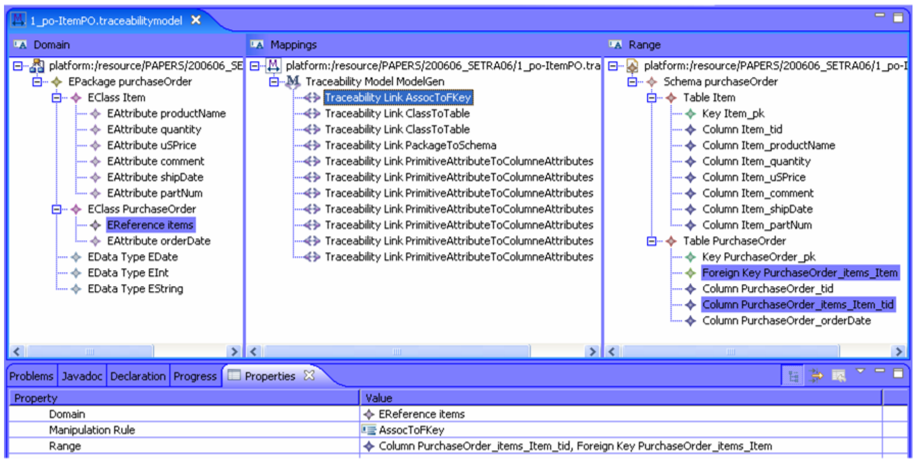
Input Parameters

Name	Type	Value
ModelGenTransfor...	Transformation	umlToRdbms
ecoreDomain	http://www.eclipse....	

Outputs

Name	Type	Value
ModelGenResult	http://es.upv.dsic....	
map0to1	TraceabilityMetamodel	

Figure 9.10: Invocation of a model transformation.



```
Domain
platform:/resource/PAPERS/200606_SE
  EPackage purchaseOrder
    EClass Item
      EAttribute productName
      EAttribute quantity
      EAttribute uSPrice
      EAttribute comment
      EAttribute shipDate
      EAttribute partNum
    EClass PurchaseOrder
      EReference items
      EAttribute orderDate
    EData Type EDate
    EData Type EInt
    EData Type EString

Mappings
platform:/resource/PAPERS/200606_SETRA06/1_po-ItemPO.tra
  Traceability Model ModelGen
    Traceability Link AssocToFKey
    Traceability Link ClassToTable
    Traceability Link ClassToTable
    Traceability Link PackageToSchema
    Traceability Link PrimitiveAttributeToColumneAttributes
    Traceability Link PrimitiveAttributeToColumneAttributes
    Traceability Link PrimitiveAttributeToColumneAttributes
    Traceability Link PrimitiveAttributeToColumneAttributes
    Traceability Link PrimitiveAttributeToColumneAttributes
    Traceability Link PrimitiveAttributeToColumneAttributes
    Traceability Link PrimitiveAttributeToColumneAttributes

Range
platform:/resource/PAPERS/200606_SETRA06/1_po-I
  Schema purchaseOrder
    Table Item
      Key Item_pk
      Column Item_tid
      Column Item_productName
      Column Item_quantity
      Column Item_uSPrice
      Column Item_comment
      Column Item_shipDate
      Column Item_partNum
    Table PurchaseOrder
      Key PurchaseOrder_pk
      Foreign Key PurchaseOrder_items_Item
      Column PurchaseOrder_tid
      Column PurchaseOrder_items_Item_tid
      Column PurchaseOrder_orderDate
```

Property	Value
Domain	EReference items
Manipulation Rule	AssocToFKey
Range	Column PurchaseOrder_items_Item_tid, Foreign Key PurchaseOrder_items_Item

Figure 9.11: Traceability editor.

9.3.4 Applications

A prototype of the MOMENT-QVT tool has been applied in the development process of several modeling frameworks. Some of these applications are becoming collaborations with other universities. We provide a brief introduction to these applications:

MOMENT Case

MOMENT Case [14] is a modeling framework that provides support for the UML2 metamodel and a relational metamodel. This prototype has been developed for the *CapGemini S.L.* company and provides the following features:

- UML2 modeling facilities, reused from the *UML2 Tools Project* [105].
- Graphical support for defining relational schemas of a given relational metamodel. An screenshot of this graphical editor is provided in Fig. 9.12.
- Automated generation of a relational schema from a UML class diagram. *MOMENT Case* provides facilities to add persistence information to a UML class diagram, indicating what elements in a class diagram will be persisted in a relational schema by means of the transformation process. This automated generation process is based on a model transformation that uses the *MOMENT-QVT* engine.
- Support for traceability, indicating what objects of the resulting relational schema have been generated from the objects of the source class diagram.
- Documentation generation from the different model definitions that may be defined in the tool: UML class diagrams, relational schemas, and traceability model definitions.
- Generation of a script in standard SQL to create the relational schema that corresponds to a model definition $\widetilde{M}$: RDBMS.

Bioinformatics

The tremendous growth of genomic sequence information combined with technological advances in the analysis of global gene expression has revolutionized research in biology and biomedicine [106]. However, the vast amounts of experimental data and associated analyses now being produced are usually persisted in heterogeneous data sources. This fact implies an urgent need for new ways of integrating this information.

A signaling pathway is constituted by a sequence of biochemical reactions by means of which a cell converts one kind of signal or stimulus into another by using genomic information. Information about processes of this kind is usually stored in relational databases, such as *TRANSPATH*. This information can be used for studying and simulating reactions that may occur in a specific signaling pathway. Some formalisms have been used for this purpose: π-calculus, ambient calculus, life sequence charts or Petri nets. An automated migration process that permits representing data from an heterogeneous data source into a

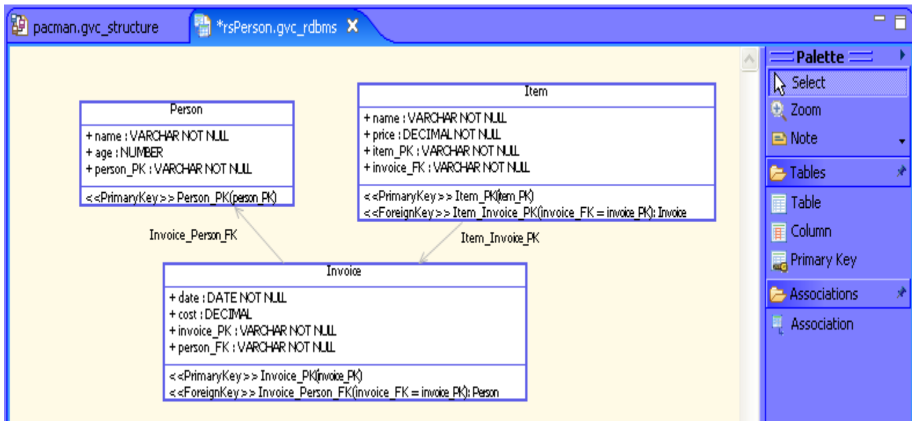

pacman.gvc_structure
*rsPerson.gvc_rdbms
Person
+ name : VARCHAR NOT NULL
+ age : NUMBER
+ person_PK : VARCHAR NOT NULL
<<PrimaryKey>> Person_PK(person_PK)
Item
+ name : VARCHAR NOT NULL
+ price : DECIMAL NOT NULL
+ item_PK : VARCHAR NOT NULL
+ invoice_FK : VARCHAR NOT NULL
<<PrimaryKey>> Item_PK(item_PK)
<<ForeignKey>> Item_Invoice_PK(invoice_FK = invoice_PK): Invoice
Invoice_Person_FK
Item_Invoice_PK
Invoice
+ date : DATE NOT NULL
+ cost : DECIMAL
+ invoice_PK : VARCHAR NOT NULL
+ person_FK : VARCHAR NOT NULL
<<PrimaryKey>> Invoice_PK(invoice_PK)
<<ForeignKey>> Invoice_Person_FK(invoice_FK = invoice_PK): Person
Palette
Select
Zoom
Note
Tables
Table
Column
Primary Key
Associations
Association

Figure 9.12: Graphical editor for the RDBMS metamodel in *MOMENT Case*.

target formalism is desirable in order to enhance analysis and simulation techniques. This migration process must face heterogeneity and interoperability challenges.

In [**?**], a migration process for representing information from the *TRANSPATH* database as coloured Petri nets has been developed. In this framework, EMF is used as the modeling framework that enhances interoperability and *MOMENT-QVT* as the underlying model transformation engine that automates the migration process. This work is based on previous experiments [107, 108], where the migration process was performed manually.

Software Metrics

Software measurement plays a fundamental role in Software Engineering [109]. Measurement can help to address some critical issues in software development and maintenance by facilitating the making of decisions. Software measurement supports planning, monitoring, controlling and evaluating the software process. In [**?**], a model-based generic framework for evaluating software metrics is presented. This framework permits the evaluation of metrics over EMF model definitions. In this approach, metrics are evaluated by means of model transformations and OCL queries, and the results of a measurement evaluation are defined in the form of a model definition $\widetilde{M}$ that conforms to a given metamodel. The *MOMENT-OCL* and *MOMENT-QVT* tools have been used to study the feasibility of the approach.

9.4 MOMENT: Model Management within the EMF.

The Model Management discipline, proposed in [110], considers model definitions as first-class citizens and provides a set of generic operators to deal with them: *Merge*, *Diff*, *ModelGen*, etc. These operators permit the direct manipulation of model definitions, instead of working on the internal representation of a model definition at a programming level. Several approaches to this discipline [111, 112] specify operators that are based on mappings to deal with model definitions. A mapping is a relationship between an object of a domain model definition and an object of a range model definition that indicates that they represent the same object in different model definitions. This means that mappings between two model definitions must be explicitly defined in order to apply an operator to them.

The MOMENT project [92] aims at the development of a model management framework that provides generic operators to manipulate MOF model definitions as first-class citizens. Some experiments have been already studied as indicated below. We informally present some of the model management operators that we use in our approach by indicating their inputs, outputs and semantics:

1. *Cross and Merge:* These operators correspond to well-defined set operations: intersection and disjoint union, respectively. Both operators receive two model definitions (A and B) as input and produce a third model definition (C). The Cross operator returns a model definition C that contains objects that participate in both the A and B input model definitions; while the *Merge* operator returns a model definition C that contains objects that belong to either the input model definition A or the input model definition B, deleting duplicated objects. Both operators also return two model definitions of links (map_{AC} and map_{BC}) that relate the objects of each input model definition to the objects of the

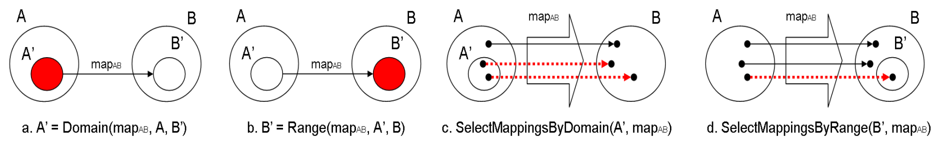

A
B
A'
B'
mapAB
a. A' = Domain(mapAB, A, B')
b. B' = Range(mapAB, A', B)
c. SelectMappingsByDomain(A', mapAB)
d. SelectMappingsByRange(B', mapAB)

Figure 9.13: Traceability Management Operators.

output model definition. Example:

$$< C, map_{AC}, map_{BC} >= Cross(A, B).$$

2. *Diff:* This operator performs the difference between two input model definitions (A and B). The difference between the two model definitions (C) is the set of objects in model definition A that does not correspond to any element in model definition B. The *Diff* operator also returns a traceability model definition that maps the objects of the model definition A to the objects of the model definition C. Example:

$$< C, map_{AC} >= Diff(A, B).$$

3. *ModelGen:* *ModelGen* performs the translation of a model definition A, which conforms to a source metamodel MMA, into a target metamodel MMB, obtaining model definition B. This operator has been presented above, in the *MOMENT-QVT* tool This operator also produces a model definition of links (map_{AB}) relating the objects of the input model definition to the objects of the generated model definition. Example:

$$< B, map_{AB} >= ModelGen_{MMA2MMB}(A).$$

Each simple operator carries out a manipulation over a set of input model definitions. An operator invokes a model transformation that is defined at the metamodel level. The semantics of this model transformation is defined axiomatically in equational logic as indicated above, and each one of its axioms is called a manipulation rule. To register the task performed over a model definition, each operator automatically produces a set of links between the objects of a source model definition and the objects of the resulting model definition. Such links are stored in traceability model definitions and are used to provide support for traceability. A mechanism to extract information from a traceability model definition, independently of the metamodel used, is needed. This mechanism consists of two kinds of operators:

- Query operators, that provide forward and backward navigation through a traceability model definition.
- Traceability management operators, to manipulate the traceability model definitions in order to automate the reasoning over traceability links. For instance, the Compose operator permits chaining traceability links in order to make implicit traceability links explicit; and the Match operator permits the inference of traceability model definitions between two model definitions. Furthermore, a traceability model definition can also be manipulated by model management operators.

We define the operators that provide navigability through a traceability model definition with the following elements: two input model definitions (A and B); a traceability model definition (map_{AB}) that relates the objects of the two input model definitions and that has been automatically produced by an operator or manually produced by a user; a model definition (A') that is a sub-model of A (i.e. A' only contains objects that also belong to A); and a model definition (B') that is a sub-model of B. The traceability operators that are considered here are:

- *Domain* and *Range:* These operators provide the backward and forward navigation through a traceability model definition, respectively. Both operators obtain a model definition as output, which is not a traceability model definition.

  The operator *Domain* takes three model definitions as input: a traceability model definition (map_{AB}), a domain model definition (A), and a range model definition (B'). The operator navigates the traceability links of the traceability model definition that have objects of B' as target objects, and returns a sub-model of A (A'), as shown in Fig. 9.13.a.

  The operator *Range* also receives three inputs: a traceability model definition (map_{AB}), a domain model definition (A'), and a range model definition (B). This operator performs the opposite task to the previous one: it navigates the traceability links that have objects of A' as domain objects and returns a sub-model of the range model definition B (B'), as shown in Fig. 9.13.b.

- *SelectMappingsByDomain* and *SelectMappingsByRange :* These operators produce a traceability model definition as output and permit selection of parts of a traceability model definition.

  The operator *SelectMappingsByDomain* receives two input model definitions: a domain model definition (A') and a traceability model definition (map_{AB}). The operator extracts the traceability links of the map_{AB} traceability model definition that have objects of the model definition A' as domain objects and returns this sub-model. The traceability links that are added to the output traceability model definition are highlighted by a dotted line in Fig. 9.13.c.

  The operator *SelectMappingsByRange* receives two input model definitions: a range model definition (B') and a traceability model definition (map_{AB}). In this case, the operator extracts the traceability links of the map_{AB} traceability model definition that have objects of the model definition B' as range objects, and returns this sub-model, as shown in Fig. 9.13.d.

In the current state of the work, the MOMENT framework permits the application of model management operators to Ecore model definitions and to RDBMS model definitions.

9.4.1 Case studies

In subsequent paragraphs we present some case studies in which we have experimented with some model management operators. Despite the algebraic manner in which the operators are used, the experiments show promising results in which the algebraic operators can be applied to model definitions that are defined in metamodeling environments like the EMF. These results validate the philosophy that has been used to define the algebraic MOF framework in this work.

Change propagation

In this case study, we use the change propagation scenario that was introduced in [111]. We illustrate it by means of a specific example shown in Fig. 9.14. We have defined the information structure of an application in a XML schema (*XSD*). To build a new application that stores the information in a relational database, we reuse the metainformation that describes the XML schema. By applying a transformation mechanism (step 1), we obtain the new relational database (*RDB*). The transformation mechanism also generates a set of links between the new generated *RDB* relational schema and the source XML schema in order to provide traceability support ($map_{XSD2RDB}$).

After obtaining a relational database from the original XML schema, we continue with the development of the new system. This may involve changes in the application and in the database (step 2), obtaining the relational schema (*RDB'*). These changes are traced and stored by the tool that manages the model manipulation or by the user directly ($map_{RDB2RDB'}$).

Once the new system is developed, changes may occur in the requirements of the system, necessitating modifications. It is easier to extend the XML schema than to modify the *RDB* database. At this point, the application of the transformation mechanism used in step 1 will discard the changes applied from *RDB* to *RDB'*.

A solution to this change propagation example can be performed by using model management operators. In our approach, traceability links are used to automate the propagation of changes that were applied to the *RDB* relational schema, for the new system C.

The problem explained in the case study can be simplified as shown in Fig. 9.15, where the $map_{XSD2RDB'}$ traceability model definition can be easily obtained from the $map_{XSD2RDB}$ and $map_{RDB2RDB'}$ traceability model definitions by means of the Compose operator. Therefore, the problem can be enunciated as follows:

*We have the following model definitions: an original XML schema (*XSD*); a XML schema (*XSD'*), which has been evolved from* XSD*; a relational database RDB', which has been generated from the XML schema* XSD *and modified afterwards; and a traceability model definition between* XSD *and RDB' ($map_{XSD2RDB'}$). The goal is to obtain a relational database from the XML schema* XSD' *that preserves the changes applied to RDB'.*

This problem can be solved by means of an operator that is defined by using other model management operators as follows:

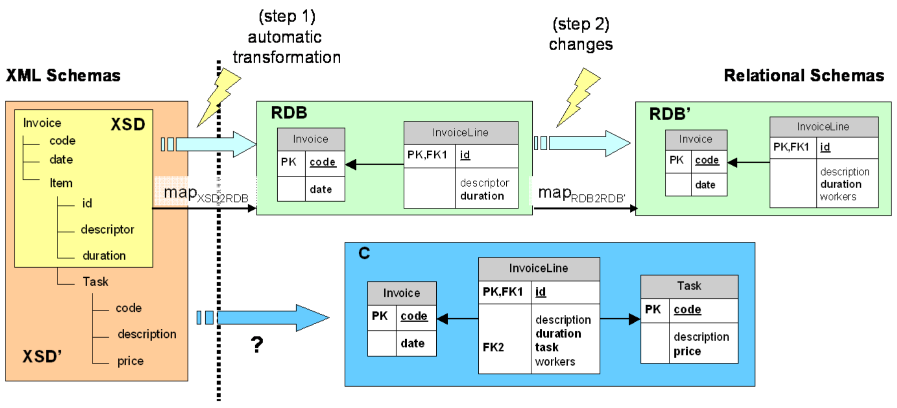


Figure 9.14: An example of change propagation.

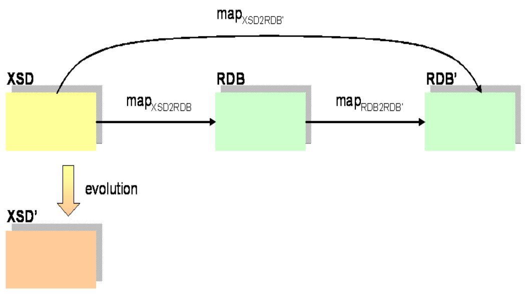


Figure 9.15: Schematization of the case study problem.

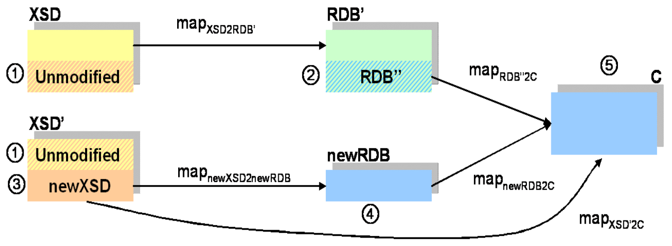


Figure 9.16: Solution of the case study problem.

$$
\begin{aligned}
&operator\,PropagateChanges(XSD, XSD', RDB', map_{XSD2RDB'}) = \\
&\quad < Unmodified, map_{XSD2Unmodified}, map_{XSD'2Unmodified} >= \\
&\qquad Cross(XSD, XSD') && (1)\\
&\quad RDB'' = Range(map_{XSD2RDB'}, Unmodified, RDB') && (2)\\
&\quad < newXSD >= Diff(XSD', Unmodified) && (3)\\
&\quad < newRDB, map_{newXSD2newRDB} >= ModelGen_{XSD2RDB}(newXSD) && (4)\\
&\quad < C, map_{RDB''2C}, map_{newRDB2C} >= Merge(RDB'', newRDB) && (5)\\
&return(C)
\end{aligned}
$$

This operator is made up of simple model management operators and the steps followed in the script are represented in Fig. 9.16. These steps are the following:

1. *Unmodified* is the part of the *XSD* model definition that remains unmodified in the *XSD'* model definition.
2. *RDB"* is the sub-model of RDB' that corresponds to the unmodified part of *XSD'*.
3. *newXSD* is the part of *XSD'* that has been added to the *XSD* model definition.
4. *newRDB* is the relational schema obtained from the translation of *newXSD* into the relational metamodel.
5. *C* is the final model definition obtained from the integration of the relational databases that we have obtained in steps 2 and 4.

If we want to add traceability support to this operator in order to generate the traceability model definition that relates the *XSD'* model definition to the new model definition (C) as well, we only have to add the next step after step 5:

$$
\begin{aligned}
&< map_{XSD'2C}, map_{mapUnmodified2C2mapXSD'2C}, map_{mapnewXSD2C2mapXSD'2C} >= \\
&\quad Merge(\\
&\qquad Compose(\\
&\qquad\quad Unmodified, \\
&\qquad\quad SelectMappingsByDomain(Unmodified, map_{XSD2RDB'}), \\
&\qquad\quad RDB'', map_{RDB''2C}, C), \\
&\qquad Compose(newXSD, map_{newXSD2newRDB}, newRDB, map_{newRDB2C}, C))
\end{aligned}
$$

This step merges two traceability model definitions: one is defined between the unmodified part of *XSD'* and C, and the other is defined between the new part of *XSD'* and C. This step merges both traceability model definitions by means of the Merge operator, in the same way that any two model definitions that belong to the same metamodel are merged. The model definition $map_{XSD'2C}$ has to be added as a return value in the script.

The resulting composite operator solves the change propagation problem of the case study independently of the metamodels involved, so that we can apply it to any combination of metamodels, instead of using the *XSD* and the relational metamodels.

Merging UML Class Diagrams

Software development methodologies based on UML propose an approach where the process is Use Case Driven [113, 114]. This means that all artifacts (including the Analysis and Design Model, its implementation and the associated test specifications) have traceability links from Use Cases. These artifacts are refined through several transformation steps. Obtaining the Analysis Model from the Use Case Model is possibly the transformation that has the least chance of achieving total automation. The Use Case Model must sacrifice precision in order to facilitate readability and validation, so that the analysis of use cases is mainly a manual activity.

When the Use Case Model has many use cases, managing traceability between each use case and the corresponding objects in the resulting class diagram can be a difficult task. In this scenario, it seems reasonable to work with each use case separately and to register its partial class diagram (which is a piece of the resulting class diagram that represents the Analysis Model). Regarding traceability, this strategy is a pragmatic solution, but when several team members work in parallel with different use cases, inconsistencies or conflicts among partial model definitions often arise, which must be solved when obtaining the integrated model.

In [10], we present a case study that illustrates how our operator *Merge* can be used effectively to deal with the required needs established above. In this work, we provide the semantics of the *Merge* operator from an algebraic point of view and we apply it to a case study: the development of part of a system for managing paper submissions that are received in a conference.

Exogenous Model Merging

In Model-Driven Engineering, model merging plays a relevant role in the maintenance and evolution of model-based software. Depending on the amount of metamodels involved in a model merging process, we can classify model merging techniques in two categories: endogenous merging, when all the model definitions to be merged conform to the same metamodel; and exogenous merging, when the model definitions to be merged conform to different metamodels.

As stated in [115], a model merging process consists of three main phases: (i) a model comparison phase, where objects of different model definitions that are equivalent are found; (ii) a consistency checking phase, where conflicts that may appear if we merge equivalent objects are identified, defining a conflict resolution strategy to eliminate them; and (iii) a merging phase, where the equivalent objects that are found in the first step are merged taking into account the conflict strategy defined in the second step.

In [13], we propose a set of model management operators that use the QVT Relations language to perform model comparison and model transformation. In a model merging process where two model definitions are involved, the comparison phase is achieved by defining relations between elements of the same metamodel. The consistency phase is solved by defining a model transformation that takes the two model definitions to be merged as input model definitions. Finally, the merging phase is performed by a generic operator that uses the QVT Relations programs defined in the previous phases. In this paper, we show how this approach can be used by providing an example of exogenous model merging, where the model definitions to be merged conform to different metamodels.

9.4.2 Pending work

The tools *MOMENT-OCL*, *MOMENT-QVT* and *MOMENT* are experimental environments where algebraic operators can be applied to EMF model definitions. In the current state of the work, these three tools are based on a previous specification of the algebraic MOF framework. Although the metamodeling philosophy is similar, the new specification that has been presented throughout Sections 5-8 enhances the reuse of Maude-based formal verification techniques. Therefore, in future work, we will base these tools on the new specification.

In addition, during the development of the tools *MOMENT-OCL*, *MOMENT-QVT* and *MOMENT*, we have kept the premise of reusing other tools as front-ends for our tools. For example, the OCL parsing facilities of the Kent Modeling Framework. In future versions of these tools, we will consider the reuse of Maude-based parsing facilities to avoid useless dependencies with complex front-ends.

In the *MOMENT-QVT* tool, only part of the semantics of the QVT Relation language is currently specified. For example, the *checkonly* semantics is still not available. In future work, we will consider specifying a complete version of the semantics of the QVT Relations language.

Finally, in the current state of the *MOMENT* framework, the model management operators are available from an algebraic point of view. However, the QVT Relations language cannot be used to refine them for a given metamodel yet, as indicated in [13]. After completing the semantics of the QVT Relations language, this task will be addressed.

9.5 Relationships to Graph Rewriting

In this section, we explain, in an illustrative, informal way, how notions of a graph rewriting system are captured by our algebraic MOF framework. In our approach, we have formalized the notions of the MOF and OCL standards in Membership Equational Logic (MEL). More specifically, a model definition $\widetilde{M}$ is algebraically represented as a graph whose structure can be traversed by using AC pattern matching, i.e., pattern matching modulo associativity and commutativity [116]. Our approach then combines the notions of several research fields:

- *Model-Driven Development:* we provide the formalization of some fundamental notions that are provided in the MOF and OCL standards.
- *Formal Metamodeling Frameworks:* we provide an algebraic executable metamodeling environment, which is plugged into the Eclipse platform, on top of the Eclipse Modeling Framework (EMF), which is used as front-end. The EMF can be regarded as an implementation of the MOF, so that the notions of the MOF standard that are formalized are mapped to the concepts of the EMF. This feature permits a graphical visualization of terms that represent model definitions $\widetilde{M}$ or metamodel definitions $\widetilde{\mathcal{M}}$. In addition, the Graphical Modeling Framework [117] can be used to attach new graphical notations to metamodel definitions.
- *Term rewriting:* in our approach we can use formal analysis techniques based on MEL and Rewriting Logic. In this section, we provide some examples.
- *Graph rewriting:* our algebraic metamodeling framework can be viewed as an executable specification of a graph rewriting system with reflection and OCL support. Graphs in our algebraic framework have an additional feature: they can be viewed as trees by considering containment properties.

In subsequent sections, we illustrate how our algebraic framework captures concepts of a graph rewriting system, and we indicate how automated analysis techniques can be applied in our framework, thanks to the underlying Rewriting Logic. The goal of this section focuses on showing applications of our framework, motivating future work.

9.5.1 Graph Rewriting Concepts in our Algebraic Framework

Graph rewriting is becoming popular as a meta-language to specify and implement visual modeling techniques [118, 119]. It may be used for parsing visual languages [120], for automated translation of visual models into code or semantics domains [121, 122], or as a semantic domain itself [123, 124]. In [125], an attributed graph is defined as a graph where nodes represent objects or data values. Edges between objects are called *links*, and edges between an object and a data value are called *attributes*. There are no edges from data vertices. The vertices of an attributed graph are typed. The corresponding types are defined in a *type graph*. In our approach, a typed attributed graph is represented by a model definition $\widetilde{M}$, as shown in Section 6, where objects constitute the nodes of the graph, object-typed properties represent directed links between objects, and value-typed properties represent attributes. A type graph is represented by a metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}}$: MOF, whose semantics is provided by the $\mathit{reflect}_{\mathrm{MOF}}$ function.

In [126], typed attributed graphs are extended with node inheritance. Subtype relationships can be defined between the nodes of a *type graph*, where types can be defined as concrete or abstract, in the sense of a concrete or an abstract object type in UML. In our approach, we consider the EMOF metamodel constructs, where inheritance can be defined between object types, and an object type can be defined as abstract too. The $\mathit{reflect}_{\mathrm{MOF}}$ function provides the algebraic semantics of the specialization relation, as shown in Section 6.

Typed attributed graphs with node inheritance permit defining model definitions $\widetilde{M}$, considering that the corresponding metamodel is defined as a type graph. However, OCL constraints can also be used in a MOF framework to define structural conditions $\widetilde{\mathcal{C}}$ over metamodel definitions $\widetilde{\mathcal{M}}$ so that model definitions $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, must satisfy them. In [127], the authors present a formal framework where type inheritance, constraints and graph transformation concepts are integrated. In particular, constraints are enforced by means of application conditions in the rules of a typed graph rewriting system, as shown in [53]. In our framework, the notion of constrained model type $(\mathcal{M}, \mathcal{C})$ is provided by the $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ theory. Values of the type $(\mathcal{M}, \mathcal{C})$ are typed attributed graphs that satisfy the OCL constraints $\mathcal{C}$. The OCL constraint

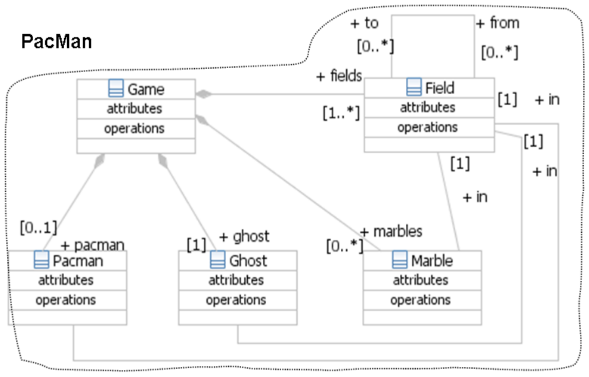


Figure 9.17: The $\widetilde{\text{PacMan}}$ metamodel.

satisfaction relation is given by a membership that ensures that a well-typed model definition satisfies the OCL constraint of the corresponding metamodel.

In [62], the authors describe a precise framework for metamodeling based on MOF concepts and graph transformation, although they do not deal with constraints. This framework considers the containment relation that can be provided in a metamodel definition. The authors discuss that containment properties are redundant notions in the MOF meta-metamodel, which can be expressed by means of package containments. However, we have chosen containment properties to define the containment relation in a metamodel definition because they can be depicted in a MOF class diagram as composite aggregations. The containment relation permits considering a model definition as a tree, enabling formal reasoning on the depth of the tree.

In addition, our framework provides a specification of the MOF Reflection Facilities, which enable the representation of typed attributed graphs independently of the corresponding type graph, as shown in Section 8. The MOF Reflection Facilities enable introspection, so that the metadata that defines the type graph can be used in the rewriting process of an instance graph. This feature is also present in the *VIATRA2* tool [128], as discussed in Section 3.

9.5.2 Graph Rewriting as Term Rewriting Modulo AC

In this section, we use a small example, borrowed from [129], to show how graph rewriting can be performed by means of term rewriting modulo AC, as introduced in [116]. The example consists in a naïve version of the PacMan game, where there is a board with fields that may contain marbles. PacMan tries to eat marbles and a ghost tries to eat PacMan. The main concepts of the game are represented as object type definitions in the metamodel definition $\widetilde{\text{PacMan}}$, shown as a MOF class diagram in Fig. 9.17. Each model definition $\widetilde{M}$, such that $\widetilde{M}$: PacMan, corresponds to a state of the game, which is denoted by a typed attributed graph. For example the initial state is defined as an object diagram in Fig. 9.18. For the sake of simplicity, we depict a state as a board, as shown in Fig. 9.19, where each field appears numbered and has its own adjacent fields. A field may have a marble. The ghost is in field 1 and PacMan is in field 10.

Roughly speaking, a graph rewriting rule ρ is a rule of the form $\rho : L \to R$, where the left-hand side L refers to the items that must be present in a model definition for an application of the rule, and the right-hand side R refers to the items that are present afterwards. Graph rewriting rules are defined in a graph grammar as *production rules*, and can be applied to a host graph by means of the double-pushout approach [130]. The movements of PacMan are defined as two graph rewriting rules: *collect*, shown in Fig. 9.20, which moves PacMan to an adjacent field with a marble and the marble disappears once Pacman moves there; and *movePM*, shown in Fig. 9.21, which moves PacMan to an adjacent field without marble. In a graph rewriting rule, a negative application condition (NAC) refers to a subgraph that must not exist in the host graph in order to apply the rule. In the graphical representation of production rules, NAC's are represented as crossed out objects.

The movements of the ghost are likewise defined by means of two production rules: *kill*, shown in Fig. 9.22, which moves the ghost to an adjacent field where PacMan is, and PacMan is killed by the application of the rule; and *moveGhost*, shown in Fig. 9.23, which moves the ghost to an adjacent field where PacMan is not located. The game is over when PacMan is deleted from the board by means of the *kill* rule.

Rewriting Logic [26] extends MEL with (possibly conditional) rewriting rules. A rewrite theory, specified in Maude as a system module, provides an executable mathematical model of a concurrent system. Mathe-

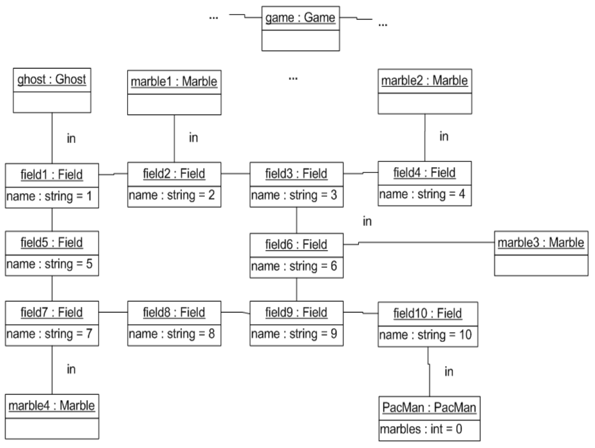


Figure 9.18: Typed attributed graph representing the initial state of a PacMan game.

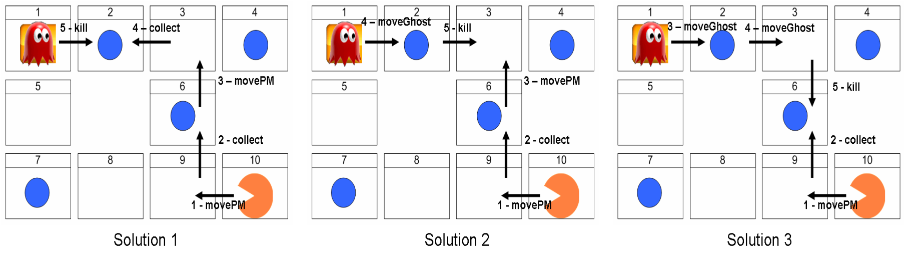


Figure 9.19: Three solutions of the PacMan game.

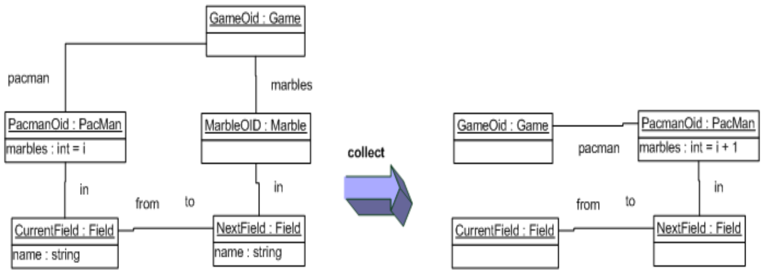


Figure 9.20: Graphical representation of the *collect* production rule of the PacMan game.

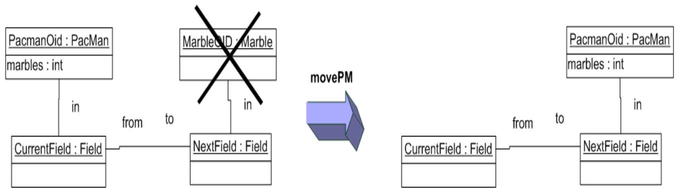


Figure 9.21: Graphical representation of the *movePM* production rule of the PacMan game.

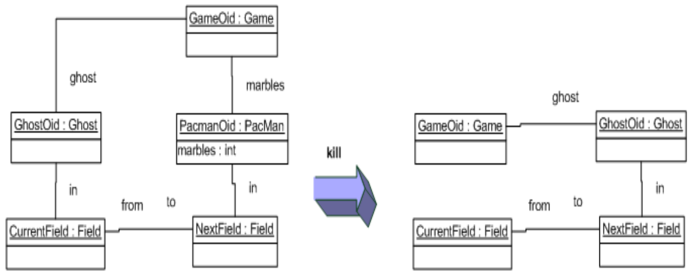


Figure 9.22: Graphical representation of the *kill* production rule of the PacMan game.

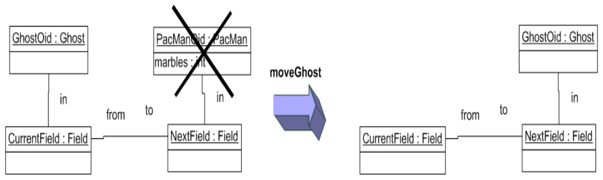


Figure 9.23: Graphical representation of the *moveGhost* production rule of the PacMan game.

matically, an unconditional rewrite rule has the form $l : t \rightarrow t'$, where t, t' are terms of the same kind, which may contain variables, and l is the label of the rule. Intuitively, a rule describes a local concurrent transition in a system. Conditional rewrite rules can have very general conditions involving equations, memberships, and other rewrites. In their Maude representation, conditional rules are declared with syntax

$$\texttt{crl [} \boxed{\mathit{Label}} \texttt{] :} \boxed{\mathit{Term\text{-}1}} \texttt{ => } \boxed{\mathit{Term\text{-}2}}$$
$$\texttt{if } \boxed{\mathit{Condition\text{-}1}} \wedge \ldots \wedge \boxed{\mathit{Condition\text{-}k}}$$
$$\texttt{[} \boxed{\mathit{StatementAttributes}} \texttt{] .}$$

A graph rewriting rule can be encoded as a rewriting rule in a natural way. Specifically, we can use the algebraic semantics that we have defined in this work to encode such graph rewriting rules. Using our algebraic semantics, the left-hand side of the rule is a term `t` of sort `ObjectCollection{MetaObject}`, such that `<< t >>` is a term of sort `ModelType{MetaObject}`, and the right-hand side is a term `t'` of sort `ObjectCollection{MetaObject}`, such that `<< t' >>` is a term of sort `ModelType{MetaObject}`. In the example, both terms represent values $\ll t \gg$ and $\ll t' \gg$ that keep a graph structure, i.e., $\ll t \gg, \ll t' \gg : \text{PacMan}^{\triangle}$. The rules of the PacMan game are then defined in Maude notation as follows:

```
mod PACMAN is
  inc META-MODEL .

  vars MatchingConf Conf : ObjectCollection{MetaEObject} .
  vars GameOid PacmanOid CurrentFieldOid NextFieldOid
    MarbleOid GhostOid : MetaOid .
  vars GameMPS PacmanMPS CurrentFieldMPS NextFieldMPS
    MarbleMPS GhostMPS : MetaPropertySet .
  vars CurrentFieldTo NextFieldFrom : OrderedSet{MetaOid} .

  crl [collect] :
    < GameOid : ecore/EObject |
        class : "pacman/Game",
        (property : "marbles" = GameMarbles:OrderedSet{MetaOid}),
        GameMPS >
    < PacmanOid : ecore/EObject |
        class : "pacman/PacMan",
        (property : "in" = CurrentFieldOid),
        (property : "marbles" = PacmanMarbles:Int),
        PacmanMPS >
    < CurrentFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "to" = CurrentFieldTo),
        CurrentFieldMPS >
    < NextFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "from" = NextFieldFrom),
        NextFieldMPS >
    < MarbleOid : ecore/EObject |
        (class : "pacman/Marble"),
        (property : "in" = NextFieldOid),
        MarbleMPS >
  =>
    < GameOid : ecore/EObject |
        (class : "pacman/Game"),
        (property : "marbles" =
            (GameMarbles:OrderedSet{MetaOid} -> excluding( MarbleOid ))),
        GameMPS >
    < PacmanOid : ecore/EObject |
        (class : "pacman/PacMan"),
        (property : "in" = NextFieldOid),
        (property : "marbles" = PacmanMarbles:Int + 1),
        PacmanMPS >
    < CurrentFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "to" = CurrentFieldTo),
        CurrentFieldMPS >
    < NextFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "from" = NextFieldFrom),
        NextFieldMPS >
  if
    (CurrentFieldTo -> includes ( NextFieldOid ))
    /\
    (NextFieldFrom -> includes ( CurrentFieldOid ))
    .


  crl [movePM] :
    < PacmanOid : ecore/EObject |
        class : "pacman/PacMan",
        (property : "in" = CurrentFieldOid),
        PacmanMPS >
    < CurrentFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "to" = CurrentFieldTo),
        CurrentFieldMPS >
    < NextFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "from" = NextFieldFrom),
        NextFieldMPS >
    Conf
  =>
    < PacmanOid : ecore/EObject |
        (class : "pacman/PacMan"),
        (property : "in" = NextFieldOid),
        PacmanMPS >
    < CurrentFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "to" = CurrentFieldTo),
        CurrentFieldMPS >
    < NextFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "from" = NextFieldFrom),
        NextFieldMPS >
    Conf
  if
    (CurrentFieldTo -> includes ( NextFieldOid ))
    /\
    (NextFieldFrom -> includes ( CurrentFieldOid ))
    /\
    (noMatchMovePM(Conf, NextFieldOid)) .


  crl [kill] :
    < GameOid : ecore/EObject |
        class : "pacman/Game",
        (property : "pacman" = PacmanOid),
        GameMPS >
    < GhostOid : ecore/EObject |
        class : "pacman/Ghost",
        (property : "in" = CurrentFieldOid),
        GhostMPS >
    < CurrentFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "to" = CurrentFieldTo),
        CurrentFieldMPS >
    < NextFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "from" = NextFieldFrom),
        NextFieldMPS >
    < PacmanOid : ecore/EObject |
        (class : "pacman/PacMan"),
        (property : "in" = NextFieldOid),
        PacmanMPS >
  =>
    < GameOid : ecore/EObject |
        (class : "pacman/Game"),
        (property : "pacman"),
        GameMPS >
    < GhostOid : ecore/EObject |
        (class : "pacman/Ghost"),
        (property : "in" = NextFieldOid),
        GhostMPS >
    < CurrentFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "to" = CurrentFieldTo),
        CurrentFieldMPS >
    < NextFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "from" = NextFieldFrom),
        NextFieldMPS >
  if
    (CurrentFieldTo -> includes ( NextFieldOid ))
    /\
    (NextFieldFrom -> includes ( CurrentFieldOid )) .


  crl [moveGhost] :
    < GameOid : ecore/EObject |
        class : "pacman/Game",
        (property : "pacman" = PacmanOid),
        GameMPS >
    < GhostOid : ecore/EObject |
        class : "pacman/Ghost",
        (property : "in" = CurrentFieldOid),
        GhostMPS >
    < CurrentFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "to" = CurrentFieldTo),
        CurrentFieldMPS >
    < NextFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "from" = NextFieldFrom),
        NextFieldMPS >
    Conf
  =>
    < GameOid : ecore/EObject |
        class : "pacman/Game",
        (property : "pacman" = PacmanOid),
        GameMPS >
    < GhostOid : ecore/EObject |
        (class : "pacman/Ghost"),
        (property : "in" = NextFieldOid),
        GhostMPS >
    < CurrentFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "to" = CurrentFieldTo),
        CurrentFieldMPS >
    < NextFieldOid : ecore/EObject |
        (class : "pacman/Field"),
        (property : "from" = NextFieldFrom),
        NextFieldMPS >
    Conf
  if
    (CurrentFieldTo -> includes ( NextFieldOid ))
    /\
    (NextFieldFrom -> includes ( CurrentFieldOid ))
    /\
    (noMatchMoveGhost(Conf, NextFieldOid)) .

endm
```

NACs can also be defined in a simple way by means of equationally-defined functions. The functions `noMatchMovePM` and `noMatchMoveGhost` define the negative application conditions for the `movePM` and `moveGhost` rules.

```
  op noMatchMovePM : Configuration{MetaEObject} MetaOid -> Bool .
  ceq noMatchMovePM( MatchingConf, NextFieldOid ) = false
  if < MarbleOid : ecore/EObject |
        (class : "pacman/Marble"),
        (property : "in" = NextFieldOid),
        MarbleMPS > Conf := MatchingConf .
  eq noMatchMovePM( MatchingConf, NextFieldOid ) = true [owise] .
  op noMatchMoveGhost : Configuration{MetaEObject} MetaOid -> Bool .
  ceq noMatchMoveGhost( MatchingConf, NextFieldOid ) = false
  if < PacmanOid : ecore/EObject |
    (class : "pacman/PacMan"),
    (property : "in" = NextFieldOid),
    MarbleMPS > Conf := MatchingConf .
  eq noMatchMoveGhost( MatchingConf, NextFieldOid) = true [owise] .
```

The above rewrite theory describes a transition system, where states are defined as model definitions $\widehat{M}$, such that $\widehat{M} : \textsc{PacMan}^{\triangle}$, and one-step transition between two states $\widehat{M}$ and $\widehat{M}'$, such that $\widehat{M}, \widehat{M}' : \textsc{PacMan}^{\triangle}$, exists if $\widehat{M} \longrightarrow^{1} \widehat{M}'$, where $\longrightarrow^{1}$ denotes the single-step application of a rewriting rule. In Maude, given an initial state, we can use the `search` command to explore the graph state that is generated for the transition system. For example, if we want to obtain three states in which the ghost has killed PacMan, we can use the command

```
search [3] in PACMAN-CONF : model =>+
    < GameOid:MetaOid : ecore/EObject |
        (class : "pacman/Game"),
        (property : "pacman"),
        GameMPS:MetaPropertySet > #
    Conf:Configuration{MetaEObject} .
```

This command returns three solutions, where each one consists in a metarepresented model definition $\widehat{M}$. The resulting solutions for the example are graphically shown in Fig. 9.19. Maude provides other commands to query the resulting state graph. For example, the command `show path` $\boxed{\textit{state}}$, where $\boxed{\textit{state}}$ is a number that identifies a specific state in the graph, provides the path between the initial state and the given state by means of rule applications. The resulting paths for the three solutions of the example are depicted in Fig. 9.19 by means of numbered arrows. A comprehensive definition of these commands, with examples, can be found in [75]. In this Section, we have shown how graph rewriting concepts can be represented in our Algebraic MOF framework. In future work, we will consider our framework as a graph rewriting system as indicated in Section 10.

Conclusions

Chapter 10

Conclusions and Future Work

This work has been inspired by the success of graph rewriting in Model-Driven Engineering and the fact that graph rewriting can be performed from an algebraic point of view in Rewriting Logic [116]. In this work, we have provided the algebraic specification of a MOF-based metamodeling framework, formalizing notions that are not clear in other approaches yet. Our work is based on the MOF and OCL standards, providing an algebraic formalization that can be reused for free, in standard-compliant frameworks, for example the Eclipse Modeling Framework (EMF) [25] and the OCL implementation of the Model Development Tools project (MDT) [100].

In our approach, we give an explicit formal representation for each of the different notions that may be involved in a metamodeling framework:

- A model definition $\widetilde{M}$ and a metamodel definition $\widetilde{\mathcal{M}}$ are syntactically represented as terms, and they are both semantically interpreted as elements in the initial algebras of the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ and $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theories, respectively.
- The model type $\mathcal{M}$ is a type model that is defined as data in a metamodel definition $\widetilde{\mathcal{M}}$. The semantics of the model type $\mathcal{M}$, $[\![\mathcal{M}]\!]_{\mathrm{MOF}}$, is given by the carrier of the sort $\mathit{ModelType}\{\mathcal{M}\}$ in the initial algebra of the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory, corresponding to all possible model definitions $\widetilde{M}$ such that $\widetilde{M} : \mathcal{M}$.
- The metamodel realization $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ is a theory that syntactically defines the model type $\mathcal{M}$ and the operators that are needed to define model definitions $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$. The notion of metamodel realization can also be used in informal approaches to refer to the program that implements the metamodel definition $\widetilde{\mathcal{M}}$, enabling the definition of models $\widetilde{M}$ as data structures.

In our framework, we distinguish between object types and model types. We say that an object is an *instance of* a given object type but that a model definition *conforms to* its model type, following the terminology used in [35, 131]. We keep this distinction to indicate that an object type is *not* a model type. Independently of the conceptual levels of the MOF framework, a model is formalized in our framework as a model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, where $\mathcal{M}$ is the corresponding model type, defined in the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory. If there is a *reflection* function

$$\mathit{reflect} : [\![\mathcal{M}]\!]_{\mathrm{MOF}} \rightsquigarrow \mathit{SpecMEL}$$

that provides the semantics of a model definition $\widetilde{M}$ as a theory $\mathit{reflect}(\widetilde{M})$, $\mathcal{M}$ is the model type, $\widetilde{M}$ is a model definition such that $\widetilde{M} : \mathcal{M}$, and $\mathit{reflect}(\widetilde{M})$ is the model theory or model realization.

In our framework, such a function is given for the MOF meta-metamodel as the $\mathit{reflect}_{\mathrm{MOF}}$ equationally-defined function. This function, together with the Object object type operations and the *reify* function, provides complete reflection support for metamodel definitions. Thus, $\widetilde{\mathrm{MOF}}$ is considered a language definition, in the sense of [39]. Note that this is a powerful notion: when a metamodel definition $\widetilde{\mathcal{M}}$ is manipulated, the corresponding theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ is also manipulated. Therefore, a metamodel definition $\widetilde{\mathcal{M}}$ cannot only be syntactically manipulated as data, but its semantics also evolve in accordance. This notion of reflection also takes into account constraint definitions by means of the *reflect* function.

The model type MOF classifies all possible metamodel definitions $\widetilde{\mathcal{M}}$, whose algebraic semantics is given by the theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$, so that model definitions that represent a system under study can be defined as $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$. In [39], Rensink discusses that MOF is not a language, because the reflection mappings are not provided in the standard. In this work, we have provided a formal version of the reflection mechanism for the metamodel definition $\widetilde{\text{MOF}}$. However, this reflection mechanism is only defined for the MOF meta-metamodel, i.e., for the domain type MOF. It cannot be reused for any metamodel definition $\widetilde{\mathcal{M}}$ such that $\widetilde{\mathcal{M}} \neq \widetilde{\text{MOF}}$. In these cases, a specific *reflect* function must be defined depending on the corresponding domain types $\mathcal{M}$ or $(\mathcal{M}, \mathcal{C})$.

In subsequent sections, we: (i) discuss some of the advantages of our metamodeling framework due to the underlying formalism, MEL; (ii) summarize the main contributions of this work; and (iii) outline some future work and open research areas.

10.1 The Advantages of Rewriting Logic and Maude

The algebraic MOF semantics *reflect* that we have explained in Sections 6-7 has two complementary aspects, one mathematical and the other operational. From the mathematical point of view, what *reflect* provides is a rigorous standard, assigning mathematical meaning to key MOF notions such as: metamodel, model, the conformance relation, and the OCL constraint satisfaction relation. There is, however, a second, very important, semantic aspect. A MEL specification (Σ, E) satisfying a few natural executability requirements can be viewed not only as a mathematical *theory*, but also as a *declarative program*, which can be efficiently executed by term rewriting in languages like Maude. In other words, the theory (Σ, E) has not only a mathematical, initial algebra semantics $T_{(\Sigma,E)}$, but also an *operational semantics*, in which execution is achieved by efficient term rewriting deduction.

What all this means for our algebraic MOF semantics *reflect* is that, since the MEL theories $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ all satisfy the executability requirements needed to use them as declarative programs, *reflect* provides not only a MOF semantics, but also a MOF *modeling environment*, in which metamodels and models can be queried and manipulated. Simply because of the executability of the modules $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, this modeling environment provides the following features *for free*:

- Data representation for metamodels, models and OCL constraints, as elements of appropriate algebraic data types.
- A reflective API to manipulate data elements such as models and metamodels in a type-agnostic way.
- Support for OCL both as a query language for the MOF framework and as a constraint language.
- In particular, this provides decision procedures for checking the conformance relation $\widetilde{M} : \mathcal{M}$, and the OCL constraint satisfaction relation $\widetilde{M} \models \mathcal{C}$.

Furthermore, the fact that, thanks to the algebraic semantics of *reflect*, all MOF concepts have a precise mathematical semantics makes now possible formal reasoning about MOF metamodels. Since *reflect* has been specified in Maude, this formal reasoning can be supported by the various formal tools in the Maude environment, including an inductive theorem prover, a model checker, and tools for checking sufficient completeness, confluence, and termination of specifications.

For example, we can use Maude's inductive theorem prover to reason about the OCL semantic consequences of a given metamodel, and about metamodel equivalence. Given a MOF metamodel specification $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ and a set $\mathcal{C}'$ of OCL constraints, we say that the constraints $\mathcal{C}'$ are a *semantic consequence* of the metamodel specification $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, written $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}) \models \mathcal{C}'$ if and only if, by definition, for each model $\widetilde{M}$, we have the implication

$$\widetilde{M} : (\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}) \;\Rightarrow\; \widetilde{M} \models \mathcal{C}'.$$

These semantic satisfaction properties can be equivalently expressed as inductive theorems in the initial model associated to $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, and can be semi-automatically proved using Maude's inductive theorem prover. In particular, we can reason in this way about the semantic equivalence between two different metamodel specifications, where, by definition, we say that $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ and $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}'})$ are *semantically equivalent*, written $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}) \cong (\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}'})$, if and only if we have

$$(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}}) \models \mathcal{C}' \;\wedge\; (\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}'}) \models \mathcal{C}.$$

On the other hand, graph rewriting can be easily obtained by means of term rewriting, as discussed in [116] and in Section 9.5. Rewriting Logic constitutes a formal framework where graph transformation concepts can be specified. If we only consider Membership Equational Logic, graph transformations can also be specified as functions, as shown in [9]. Furthermore, automated formal verification techniques for term rewriting can then be reused for graph rewriting.

10.2 Summary of Contributions

From a theoretical point of view, our work constitutes an algebraic metamodeling framework where the following notions, which are present in a MOF metamodeling approach, have been defined in a precise way:

- the MOF meta-metamodel realization as the $\mathit{reflect}(\widetilde{\mathrm{MOF}}, \emptyset)$ theory;
- $\widetilde{\mathcal{M}}$: metamodel definition, or model type definition;
- $\mathit{reflect}(\widetilde{\mathcal{M}}, \emptyset)$: metamodel realization;
- $\mathcal{M}$: metamodel (in the MEL theory $\mathit{reflect}(\widetilde{\mathcal{M}}, \emptyset)$, $\mathcal{M}$ is represented by the $\mathit{ModelType}\{\mathcal{M}\}$ sort);
- $\widetilde{M} : \mathcal{M}$: structural conformance relation;
- $[\![\mathcal{M}]\!]_{\mathrm{MOF}}$: semantics of the model type $\mathcal{M}$;
- $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$: metamodel specification definition;
- $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$: metamodel specification realization;
- $(\mathcal{M}, \mathcal{C})$: metamodel specification (in the MEL theory $\mathit{reflect}(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$, $(\mathcal{M}, \mathcal{C})$ is represented by the $\mathit{ConsistentModelType}\{\mathcal{M}\}$ sort);
- $\widetilde{M} \models \mathcal{C}$: OCL constraint satisfaction relation;
- $\widetilde{M} : (\mathcal{M}, \mathcal{C})$: constrained conformance relation; and
- $[\![(\mathcal{M}, \mathcal{C})]\!]_{\mathrm{MOF}}$: semantics of the consistent model type $(\mathcal{M}, \mathcal{C})$;
- the MOF OBJECT type and its operations.

In addition, we have aligned the system of types of the MOF and OCL metamodels so that OCL expressions can be algebraically defined with introspection facilities, and properties of object types can be defined in a metamodel definition $\widetilde{\mathcal{M}}$ by means of OCL collection types. From an executability point of view, we have extended the existing support for object-oriented programming in Maude, enabling *model-oriented programming* by:

- introducing notions of Model-Driven Development;
- introducing notions of the MOF and OCL metamodels, such as enumeration types, object-typed properties, containment properties, and OCL collection types;
- providing support for querying the graph and tree views of a model definition;
- providing support for structural OCL constraints;
- introducing MOF-based reflection, which provides the algebraic semantics of a metamodel definition and introspection facilities; and
- providing an Eclipse plugin that maps EMF models to terms that represent a model definition $\widetilde{M}$.

10.3 Future Work

In this work, we have given a detailed definition of the *reflect* function that permits projecting a metamodel specification definition $(\widetilde{\mathcal{M}}, \widetilde{\mathcal{C}})$ as a theory, enabling the definition $\widetilde{M}$ of models that both conform to $\mathcal{M}$, $\widetilde{M} : \mathcal{M}$, and satisfy the OCL constraints $\widetilde{\mathcal{C}}$, $\widetilde{M} \models \mathcal{C}$.

The algebraic MOF framework has been specified in Maude and has been integrated into the EMF. The specification includes the infrastructure of MEL theories that has been presented throughout the Sections 6-8. As indicated in Section 9, in the current implementation only metamodel realizations that provide support for the structural conformance relation are supported by means of the $\mathit{reflect}_{\mathrm{MOF}}$ function. In future work, we will consider the complete definition of the *reflect* function, providing support for the constrained conformance relation. This function has been mathematically defined in this work based on the experience in a previous prototype for OCL constraint validation [11]. However, the support for OCL expressions is already available in the algebraic framework, although the concrete syntax for the OCL language is still not supported. Another choice to provide support for the OCL constraint satisfaction relation consists in defining the syntax and the semantics of the OCL language as indicated in the *Rewriting Logic Semantics* project [101, 102]. We will also consider this second choice.

Furthermore, introspection is already available in algebraic OCL expressions in our algebraic MOF framework. However, the concrete syntax of OCL does not support the use of the OBJECT object type operations in an OCL expression. To support this feature, the OBJECT object type has to be included in the OCL standard library, enabling introspection facilities for OCL.

10.4 Open Research Areas

The metamodeling framework that has been presented in this work opens several research areas, which involve either extending the current framework with new notions or using it as the kernel of a model management framework. In this section, we outline some of these open research areas.

10.4.1 Metamodeling Aspects

A metamodel definition $\widetilde{\mathcal{M}}$ constitutes a reusable container of type definitions $\widetilde{T}$ that can be enriched or extended by other metamodel definitions. For example, the metamodel definition $\widetilde{OCL}$ imports the metamodel definition $\widetilde{\text{MOF}}$ to enable the use of user-defined types $\widetilde{T}$ of a metamodel definition $\widetilde{\mathcal{M}} : \text{MOF}$ in specific OCL expressions $\widetilde{c} : \text{OCL}$.

The metamodel extension mechanism can be studied in two different cases: (i) when metamodels are given as data $\widetilde{\mathcal{M}}$, and (ii) when metamodels are given as semantically-defined entities $\mathcal{M}$. At present, the MOF standard only provides extension metamodel mechanisms on the data version $\widetilde{\mathcal{M}}$ of metamodels. When a metamodel definition $\widetilde{\mathcal{M}}$ is realized as a program in a specific MOF-based modeling environment, these mechanisms depend on the environment-specific semantics that is given by code generation from a MOF metamodel definition $\widetilde{\mathcal{M}}$ to a target (OO) programming language. In these cases, the semantics that is provided for $\mathcal{M}$ is not formal, and depends on specific implementation details. Furthermore, although OCL constraints can be used to define well-formed requirements in a MOF metamodel definition $\widetilde{\mathcal{M}}$, these are not taken into account explicitly in the metamodel extension mechanism. A formal definition of an extension relation between model types $\mathcal{M}$ and/or between constrained model types $(\mathcal{M}, \mathcal{C})$ would allow supporting polymorphism in functions that are typed with such model types, like model transformations. In [37], the authors provide a formal extension relation between metamodels discussing its advantages.

On the other hand, some metamodeling frameworks, like the Eclipse Modeling Framework, also provide support for defining parameterized object types in a metamodel definition. Providing the formal semantics of this feature would allow a more expressive metamodeling framework. Maude already provides support for theories that are parameterized with objects [75], so that the formalization of this notion is also feasible. EMF also provides support for XML, so that an XML schema can be automatically imported as a metamodel definition. However, the Ecore meta-metamodel provides constructs to deal with XML features that are not considered yet in our framework. This second feature would allow the direct, automated formalization of a broader set of metamodels, provided as XML schemas.

10.4.2 Precise Model Transformation and Model Management

Our metamodeling framework permits using model definitions $\widetilde{M}$ as first-class citizens, rising the level of abstraction of model-based tasks, where the internals of a specific model remain hidden. For example, a model transformation that is defined at level M2 between a source metamodel specification definition $(\widetilde{\mathcal{A}}, \widetilde{\mathcal{C}_{\mathcal{A}}})$ and a target metamodel specification definition $(\widetilde{\mathcal{B}}, \widetilde{\mathcal{C}_{\mathcal{B}}})$ can be mathematically defined in our framework as a function

$$f : [\![(\mathcal{A}, \mathcal{C}_{\mathcal{A}})]\!]_{\text{MOF}} \to [\![(\mathcal{B}, \mathcal{C}_{\mathcal{B}})]\!]_{\text{MOF}}.$$

Given a model definition $\widetilde{M} : (\mathcal{A}, \mathcal{C}_{\mathcal{A}})$, we can then use the model $f(\widetilde{M})$, where $f(\widetilde{M}) : (\mathcal{B}, \mathcal{C}_{\mathcal{B}})$ without any need for knowing the specific objects that constitute either $\widetilde{M}$ or $f(\widetilde{M})$. Note that, in addition, the sets $\mathcal{C}_{\mathcal{A}}$ and $\mathcal{C}_{\mathcal{B}}$ of OCL constraints are implicitly taken into account without any need for performing additional checking tasks.

10.4.3 Model-based Formal Verification Techniques

In our approach, we are using Maude's implementation of Rewriting Logic, so that we can reuse all Maude-based facilities for automated formal verification. A rewrite theory, specified in Maude as a system module, provides an executable mathematical model of a concurrent system. Mathematically, an unconditional rewrite rule has the form $l : t \to t'$, where t, t' are terms of the same kind, which may contain variables, and l is the label of the rule. Intuitively, a rule describes a local concurrent transition in a system, where the terms t and t' form part of different states. As indicated in [75], the *search* command that is provided in Maude can be used to apply (bounded) model checking of invariants. Such invariants can be defined as OCL constraints in our MOF framework. In addition, [132] provides a simple method to define finite-state abstractions of a state space, i.e., an appropriate quotient of the original system whose set of reachable states is finite, by just adding equations. We leave a comparison of these techniques with other graph rewriting-based model checking techniques for future work.

10.4.4 Bridging the Gap Between Grammarware and Modelware

(Forward) Model-Driven Engineering (MDE) [21] increases the level of abstraction of software artifacts in a software development process, enhancing interoperability and productivity. A huge effort is being done in applying MDE to industry practices through initiatives such as OMG's Model-Driven Architecture (MDA) [18] and the Modelware Project [133]. Despite these efforts, industry may remain code-centric for a long time, as stated in [134], since large industrial software products are still mostly made out of raw software items using legacy technology. Therefore, Model-Driven reverse Engineering (MDrE) processes are needed to enable the use of such legacy software within model-based software artifacts.

Grammarware comprises grammars and all grammar-dependent software, i.e., software artifacts that directly involve grammar knowledge, playing a key role in Reverse Model-Driven Engineering. In [135], a survey of techniques that are applied to grammar-based languages is provided. Grammarware involves both general purpose languages (GPLs), such as C#, Java or XML (among many others), and Domain Specific Languages (DSLs) [136]. DSLs are languages tailored to a specific application domain. They increase the expressiveness and ease of use compared with GPLs in the corresponding domain of application. There are several proposals to bridge model-based languages and grammar-based languages so that MDrE processes can be made feasible [137, 138, 139, 140, 141].

In MEL, a context-free grammar G can be represented as an order-sorted signature Σ_G with mix-fix syntax operators [142]. For example, given a production rule,

$$<A> ::= <C>\ \ bc\ \ <A>\ \ b\ \ <A>\ \ a$$

with a, b, c terminal symbols and A, C non-terminals, we obtain a corresponding operator declaration

$$\_bc\_b\_a : \ C\ A\ A\ \rightarrow\ A.$$

Due to the reflective features of MEL, bridging grammar-based languages and model-based languages is feasible in our algebraic metamodeling framework by means of the following steps:

From grammar to metamodel ($G2MM$): generation of one and only one metamodel definition $\widetilde{\mathcal{M}}$ for a context-free grammar G that defines the concrete syntax of a language. Let *Module* be the sort of metarepresented MEL theories in MEL, *G2MM* can be represented as an equationally-defined function:

$$\overline{G2MM} : Module \longrightarrow ConsistentModelType\{MOF\}.$$

From metamodel to grammar ($MM2G$): generation of a context-free grammar from a metamodel definition $\widetilde{\mathcal{M}}$ that defines the abstract syntax of a language. *MM2G* can be represented as an equationally-defined function:

$$\overline{MM2G} : ConsistentModelType\{MOF\} \longrightarrow Module.$$

From program to model ($P2M$): generation of one and only one model definition $\widetilde{M}$, such that $\widetilde{M} : \mathcal{M}$, from a given program P, which is based on the grammar G. The generated model definition $\widetilde{M}$ represents the abstract syntax tree of the corresponding program P. Let *Term* be the sort of metarepresented terms in MEL, *M2P* can be represented as an equationally-defined function:

$$\overline{P2M} : Term \longrightarrow ConsistentModelType\{\mathcal{M}\}.$$

From model to program ($M2P$): generation of a program P based on a grammar G from a model definition $\widetilde{M}$ that conforms to the corresponding model type $\mathcal{M}$. This generation process constitutes a backward mechanism from a model to code (forwards in the sense of MDA), which can be equationally-defined as a function:

$$\overline{M2P} : ConsistentModelType\{\mathcal{M}\} \longrightarrow Term.$$

Functions of this kind, which relate grammars to metamodels and programs to models, provide formal support to MDrE processes. Model-driven processes that can be supported by this approach include:

Automated generation of model-based languages. From any kind of language that is based on a context-free grammar G, either a general purpose language (such as Java) or a DSL, the function *G2MM* generates a MOF metamodel definition $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}} = \overline{G2MM}(\overline{\Sigma_G})$. Through the function *P2M*, a program P that is well-formed in the language (L_G), which is defined by G, corresponds to exactly one model definition $\widetilde{M}$ so that $\widetilde{M} : \mathcal{M}$. Both the *G2MM* and *P2M* functions provide support for a reverse MDE process allowing the recovery of (possibly legacy) programs into models and the automated generation of documentation for legacy code. Furthermore, visual concrete syntax can be attached to elements of the metamodel by means of MOF-based technology, such as the Graphical Modeling Framework, endowing formal languages with visual facilities.

Formal semantic definition of EMOF-based languages. Both syntax and semantics of a programming language L_G based on a grammar G can be defined in rewriting logic, as described in [101, 102]. By means of the function $M2P$, a model definition $\widetilde{M}$ that conforms to the generated metamodel definition $\overline{\mathit{G2MM}(\overline{\Sigma_G})}$ correspond to exactly one program P that is well-formed in the language L_G. Thus, the program P can be interpreted by means of the semantics defined for L_G. The operational semantics of the rewriting logic definition of a grammar-based language, in a rewriting logic implementation like Maude, together with the bridges that have been introduced in this paper, not only provide an efficient interpreter for models that conform to MOF metamodels, but also enable the application of powerful program analysis techniques, such as model checking, to such models.

Automated code generation for models. A lightweight process can be easily obtained from the previous one by only defining the syntax of the language L_G as an order-sorted signature. Since the function $G2MM$ does not take into account the rewriting logic semantic definition of L_G, a well-formed MOF metamodel definition $\overline{\mathit{G2MM}(\overline{\Sigma_G})}$ is obtained. In this case, the function $M2P$ acts as an automated code generator, since the term that is generated from a model that conforms to the metamodel definition $\overline{\mathit{G2MM}(\overline{\Sigma_G})}$ represents the concrete syntax of a program in L_G.

Formal Reasoning over PIMs. Model transformations can be used to provide support for a MDE process, where a Platform Independent Model (PIM), such as a UML model, can be transformed into a Platform Specific Model (PSM), such as a Java model. If the platform specific metamodel corresponds to a grammar-based language L_G, whose semantics has been defined in rewriting logic, the resulting PSM is directly executable. Furthermore, if the rewriting logic definition of a model transformation language is also provided, the formal semantics of the PIM model could be defined by composing the semantics of the PIM-to-PSM transformation definition and the rewriting logic definition of the language L_G. This fact will enable the application of program analysis techniques to PIM models, enhancing the formal specification of software artifacts in the early stages of a model-driven development process.

Round-trip. The functions $P2M$ and $M2P$ provide round-trip support, so that manual changes to code can be directly reflected into the corresponding PSM, and changes to a PSM can be automatically reflected into changes to the corresponding code. Model transformation engines with support for traceability can also be used to complete a round-trip process from a PIM to code. Indeed, model management frameworks provide support for tasks of this kind by means of generic, composable operators that can manipulate and query models. In [7], an example of how these operators can be used to solve a change propagation scenario is provided.

References

[1] Boronat, A., Carsí, J.A., Ramos, I.: Una plataforma semántica para la gestión de modelos. In Pimentel, E., Brisaboa, N.R., Gómez, J., eds.: JISBD. (2003) 167–176

[2] Boronat, A., Pérez, J., Carsí, J.A., Ramos, I.: Two experiences in software dynamics. Journal of Universal Computer Science **10**(4) (2004) 428–453

[3] Boronat, A., Ramos, I., Carsí, J.A.: Automatic model generation in model management. In Das, G., Gulati, V.P., eds.: CIT. Volume 3356 of Lecture Notes in Computer Science., Springer (2004) 326–335

[4] Boronat, A., Carsí, J.A., Ramos, I., Pedrós, J.: An Approach for Cross-Model Semantic Transformation on the .NET Framework. In: .NET Technologies2005 conference proceedings. Plzen, Czech Republic. (2005)

[5] Boronat, A., Carsí, J.A., Ramos, I.: Automatic reengineering in mda using rewriting logic as transformation engine. In: CSMR, IEEE Computer Society (2005) 228–231

[6] Boronat, A., Carsí, J.A., Ramos, I.: An algebraic baseline for automatic transformations in mda. Electr. Notes Theor. Comput. Sci. **127**(3) (2005) 31–47

[7] Boronat, A., Carsí, J.A., Ramos, I.: Automatic support for traceability in a generic model management framework. In Hartman, A., Kreische, D., eds.: Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005. Volume 3748 of Lecture Notes in Computer Science., Springer (2005) 316–330

[8] Boronat, A., Iborra, J., Carsí, J.A., Ramos, I., Gómez, A.: Del método formal a la aplicación industrial en gestión de modelos: Maude aplicado a eclipse modeling framework. In Álvarez, J.A.T., Núñez, J.H., eds.: JISBD, Thomson (2005) 253–258

[9] Boronat, A., Carsí, J.A., Ramos, I.: Algebraic specification of a model transformation engine. In Baresi, L., Heckel, R., eds.: FASE. Volume 3922 of Lecture Notes in Computer Science., Springer (2006) 262–277

[10] Boronat, A., Carsí, J.A., Ramos, I., Letelier, P.: Formal model merging applied to class diagram integration. Electr. Notes Theor. Comput. Sci. **166** (2007) 5–26

[11] Boronat, A., Oriente, J., Gómez, A., Ramos, I., Carsí, J.A.: An algebraic specification of generic ocl queries within the eclipse modeling framework. In Rensink, A., Warmer, J., eds.: ECMDA-FA. Volume 4066 of Lecture Notes in Computer Science., Springer (2006) 316–330

[12] Boronat, A., , Ramos, I., Carsí, J.A.: Definition of ocl 2.0 operational semantics by means of a parameterized algebraic specification. In Ramos, I., Carsí, J.A., Boronat, A., eds.: 1st International Workshop on Algebraic Foundations for OCL and Applications, Technical University of Valencia (2006)

[13] Boronat, A., Carsí, J.A., Ramos, I.: Exogenous model merging by means of model management operators. Electronic Communications of the EASST **3** (2006) `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/8`.

[14] Gómez, A., Boronat, A., Carsí, J.A., Ramos, I.: MOMENT CASE: Un prototipo de herramienta CASE (Demo) (to appear - in Spanish). [143]

[15] Mora, B., García, F., Ruiz, F., Piattini, M., Boronat, A., Gómez, A., Carsí, J.A., Ramos, I.: Marco de Trabajo basado en MDA para la Medicin Genérica del Software (to appear - in Spanish). [143]

[16] Gómez, A., Boronat, A., Carsí, J.A., Ramos, I.: Recuperación y procesado de datos biológicos mediante Ingeniería Dirigida por Modelos (to appear - in Spanish). [143]

[17] Sztipanovits, J., Karsai, G.: Model-integrated computing. Computer **30**(4) (1997) 110–111

[18] Object Management Group: MDA Guide Version 1.0.1. (2003) `http://www.omg.org/docs/omg/03-06-01.pdf`.

[19] Frankel, D.: Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley and Sons, Inc., New York, NY, USA (2002)

[20] Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley and Sons (2004)

[21] Schmidt, D.C.: Model-driven engineering. IEEE Computer **39**(2) (2006)

[22] Object Management Group: Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01) (2006) `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`.

[23] Object Management Group: MOF 2.0 QVT final adopted specification (ptc/05-11-01) (2005) `http://www.omg.org/cgi-bin/doc?ptc/2005-11-01`.

[24] Object Management Group: OCL 2.0 Specification (2006) `http://www.omg.org/cgi-bin/doc?formal/2006-05-01`.

[25] Eclipse Organization: The eclipse modeling framework (2007) `http://www.eclipse.org/emf/`.

[26] Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1) (1992) 73–155

[27] Meseguer, J.: Membership algebra as a logical framework for equational specification. In Parisi-Presicce, F., ed.: Proc. WADT'97, Springer LNCS 1376 (1998) 18–61

[28] NetBeans: Netbeans metadata repository (2007) `http://mdr.netbeans.org/`.

[29] Microsoft Corp.: The DSL tools (2007) `http://msdn.microsoft.com/vstudio/DSLTools/`.

[30] Metacase, Corp.: Metaedit web site (2007) `http://www.metacase.com/`.

[31] Warmer, J., Kleppe, A.: The Object Constraint Language, Second Edition, Getting Your Models Ready for MDA. Addison-Wesley (2004)

[32] Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14 (2002)

[33] Java Community Process: The Java Metadata Interface (JMI) Specification (JSR 40) (2002) `http://www.jcp.org/en/jsr/detail?id=40`.

[34] Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20**(5) (2003) 19–25

[35] Bezivin, J.: On the unification power of models. Software and System Modeling (SoSym) **4**(2) (2005a) 171–188

[36] Poernomo, I.: The meta-object facility typed. [144] 1845–1849

[37] Steel, J., Jézéquel, J.M.: On model typing. Journal of Software and Systems Modeling (SoSyM) (2006)

[38] Seidewitz, E.: What models mean. Software, IEEE **20**(5) (2003) 26–32

[39] Rensink, A.: Subjects, models, languages, transformations. In Bézivin, J., Heckel, R., eds.: Language Engineering for Model-Driven Software Development. Volume 04101 of Dagstuhl Seminar Proceedings., Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2004)

[40] Kuhne, T.: Matters of (meta-) modeling. Software and Systems Modeling (SoSyM) **5** (December 2006) 369–385(17)

[41] Hesse, W.: More matters on (meta-)modelling: remarks on thomas kuhnes matters. Software and Systems Modeling (SoSyM) **5**(4) (2006) 387–394

[42] Kuhne, T.: Clarifying matters of (meta-) modeling: an authors reply. Software and Systems Modeling (SoSyM) **5**(4) (2006) 395–401

[43] Ludewig, J.: Models in software engineering - an introduction. Inform., Forsch. Entwickl. **18**(3-4) (2004) 105–112

[44] Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. [145] 19–33

[45] The MOVA group: The MOVA tool: a validation tool for UML (2006) `http://maude.sip.ucm.es/mova/`.

[46] Basin, D.A., Doser, J., Lodderstedt, T.: Model driven security: From UML models to access control infrastructures. ACM Trans. Softw. Eng. Methodol. **15**(1) (2006) 39–91

[47] Clavel, M., Egea, M.: ITP/OCL: A rewriting-based validation tool for uml+ocl static class diagrams. In Johnson, M., Vene, V., eds.: AMAST. Volume 4019 of Lecture Notes in Computer Science., Springer (2006) 368–373

[48] Clavel, M., Egea, M.: Equational specification of uml+ocl static class diagrams (2006) `http://maude.sip.ucm.es/~clavel/pubs/clavel-egea06a.pdf`.

[49] Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varró, D., , Varró-Gyapay, S.: Model transformation by graph transformation: A comparative study. In: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica (2005)

[50] (AGG Homepage) `http://tfs.cs.tu-berlin.de/agg/`.

[51] Bardohl, R., Ehrig, H., de Lara, J., Taentzer, G.: Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. [146] 214–228

[52] Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. [147] 161–177

[53] Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Theory of constraints and application conditions: From graphs to high-level structures. Fundam. Inform. **74**(1) (2006) 135–166

[54] Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., Varró-Gyapay, S.: Termination criteria for model transformation. [148] 49–63

[55] Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: Termination of high-level replacement units with application to model transformation. Electr. Notes Theor. Comput. Sci. **127**(4) (2005) 71–86

[56] de Lara, J., Vangheluwe, H.: Atom^3: A tool for multi-formalism and meta-modelling. In Kutsche, R.D., Weber, H., eds.: FASE. Volume 2306 of Lecture Notes in Computer Science., Springer (2002) 174–188

[57] Rozenberg, G., ed.: Handbook of graph grammars and computing by graph transformation: volume I. foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1997)

[58] Guerra, E., de Lara, J.: Event-driven grammars: Towards the integration of meta-modelling and graph transformation. [147] 54–69

[59] Guerra, E., Díaz, P., de Lara, J.: A formal approach to the generation of visual language environments supporting multiple views. In: VL/HCC, IEEE Computer Society (2005) 284–286

[60] Schürr, A.: Specification of graph translators with triple graph grammars. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: WG. Volume 903 of Lecture Notes in Computer Science., Springer (1994) 151–163

[61] Budapest University of Technology and Economics: Viatra2 (2007) `http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html`.

[62] Varró, D., Pataricza, A.: Vpm: A visual, precise and multilevel metamodeling framework for describing mathematical domains and uml (the mathematics of metamodeling is metamodeling mathematics). Software and System Modeling **2**(3) (2003) 187–210

[63] Varró, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In Baar, T., Strohmeier, A., Moreira, A., Mellor, S., eds.: Proc. UML 2004: 7th International Conference on the Unified Modeling Language. Volume 3273 of LNCS., Lisbon, Portugal, Springer (2004) 290–304

[64] Heckel, R.: Compositional verification of reactive systems specified by graph transformation. In: FASE. (1998) 138–153

[65] Rensink, A., Schmidt, Á., Varró, D.: Model checking graph transformations: A comparison of two approaches. [147] 226–241

[66] Schmidt, Á., Varró, D.: Checkvml: A tool for model checking visual modeling languages. In Stevens, P., Whittle, J., Booch, G., eds.: UML. Volume 2863 of Lecture Notes in Computer Science., Springer (2003) 92–95

[67] Varró, D.: Towards symbolic analysis of visual modeling languages. Electr. Notes Theor. Comput. Sci. **72**(3) (2003)

[68] Varró, D.: Automated formal verification of visual modeling languages by model checking. Software and System Modeling **3**(2) (2004) 85–113

[69] Holzmann, G.J.: The model checker spin. IEEE Trans. Software Eng. **23**(5) (1997) 279–295

[70] Dániel Varró, András Balogh, A.P.: The VIATRA2 Transformation Framework: Model transformation by Graph Transformation. In: Eclipse Modeling Symposium. (2006) `http://www.eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium7_VIATRA2TransformationFramework.pdf`.

[71] Rensink, A.: Groove: GRaphs for Object-Oriented VErification (2007) `http://janus.cs.utwente.nl/~groove/wordpress/groove-home/`.

[72] Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of visual editors as eclipse plug-ins. In Redmiles, D.F., Ellman, T., Zisman, A., eds.: ASE, ACM (2005) 134–143

[73] Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted ocl constraints into graph constraints for generating meta model instances by graph grammars. In: 5th International Workshop on Graph Transformation and Visual Modeling Techniques. GT-VMT 2006. April 1 - 2 2006, Vienna, Austria. In proceedings. (To Appear). (2006)

[74] Ehrig, K., Winkelmann, J.: Model transformation from visualocl to ocl using graph transformation. Electr. Notes Theor. Comput. Sci. **152** (2006) 23–37

[75] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude: a High-Performance Logical Framework. Springer (2007)

[76] Meseguer, J.: A logical theory of concurrent objects and its realization in the Maude language. In Agha, G., Wegner, P., Yonezawa, A., eds.: Research Directions in Concurrent Object-Oriented Programming. MIT Press (1993) 314–390

[77] Fernández, J.L., Toval, A.: Seamless Formalizing the UML Semantics through Metamodels. Unified Modeling Language: Systems Analysis, Design, and Development Issues. Idea Group Publishing (2001)

[78] Fernández, J.L., Toval, A.: Can intuition become rigorous? foundations for uml model verification tools. In: International Symposium on Software Reliability Engineering (ISSRE 2000), IEEE (2000) San Jose, California, USA.

[79] Alemán, J.L.F.: A formalization proposal of the UML four-layered architecture. PhD thesis, Murcia University (2002) (In Spanish).

[80] Toval, A., Fernández, J.L.: Formally modeling uml and its evolution: a holistic approach. Kluwer Academic Publishers (2000) FMOODS'00, Formal Methods for Open Object-Based Distributed Systems. Stanford, California, USA.

[81] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.: Maude: specification and programming in rewriting logic. Theoretical Computer Science **285** (2002) 187–243

[82] Clavel, M., Meseguer, J., Palomino, M.: Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. In Gadducci, F., Montanari, U., eds.: Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications. Volume 71., ENTCS, Elsevier (2002)

[83] Clavel, M.: Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications. CSLI Publications (2000)

[84] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J.: Metalevel computation in Maude. Volume 15., Elsevier (1998) 3–24 `http://www.elsevier.nl/locate/entcs/volume15.html`.

[85] Burstall, R.M., Goguen, J.A.: The semantics of clear, a specification language. In Bjørner, D., ed.: Abstract Software Specifications. Volume 86 of Lecture Notes in Computer Science., Springer (1979) 292–332

[86] Goguen, J.A., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.P.: Introducing OBJ. In Goguen, J.A., Malcolm, G., eds.: Software Engineering with OBJ: Algebraic Specification in Action. Advances in Formal Methods. Kluwer Academic Publishers (2000) 3–167

[87] Poernomo, I.: The meta-object facility typed. [144] 1845–1849

[88] Brucker, A.D., Wolff, B.: The HOL-OCL book. Technical Report 525, ETH Zürich (2006)

[89] Object Management Group: UML 2.0 superstructure specification (formal/05-07-04) (2004) `http://www.omg.org/cgi-bin/doc?formal/05-07-04`.

[90] Object Management Group: UML 2.0 infrastructure specification (formal/05-07-05) (2004) `http://www.omg.org/cgi-bin/doc?formal/05-07-05`.

[91] Gerber, A., Raymond, K.: Mof to emf: there and back again. In: eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange, New York, NY, USA, ACM Press (2003) 60–64

[92] The ISSI Research Group: (The MOMENT Project) `http://moment.dsic.upv.es`.

[93] Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14 (2002)

[94] Baar, T.: Non-deterministic constructs in ocl - what does any() mean. In Prinz, A., Reed, R., Reed, J., eds.: SDL Forum. Volume 3530 of Lecture Notes in Computer Science., Springer (2005) 32–46

[95] Smith, B.: Informal proceedings first workshop on reflection and metalevel architectures in object-oriented programming, oopsla/ecoop'90 (1990)

[96] Bobrow, D.G., Gabriel, R.P., White, J.L.: Clos in context: the shape of the design space. (1993) 29–61

[97] Demers, F.N., Malenfant, J.: Reflection in logic, functional and object-oriented programming: a short comparative study. In: Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI. (1995)

[98] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework: A Developer's Guide. Pearson Education (2003)

[99] Akehurst, D., Patrascoiu, O., Smith, R.: Kent Modeling Framework (2006) `http://www.cs.kent.ac.uk/projects/ocl/`.

[100] Eclipse Organization: Model development tools (2007) `http://www.eclipse.org/modeling/mdt/`.

[101] Meseguer, J., Rosu, G.: The rewriting logic semantics project. Electr. Notes Theor. Comput. Sci. **156**(1) (2006) 27–56

[102] Meseguer, J., Rosu, G.: Rewriting logic semantics: From language specifications to formal analysis tools. In Basin, D.A., Rusinowitch, M., eds.: IJCAR. Volume 3097 of Lecture Notes in Computer Science., Springer (2004) 1–44

[103] Bézivin, J., Hillairet, G., Jouault, F., Kurtev, I., Piers, W.: Bridging the ms/dsl tools and the eclipse modeling framework. In: OOPSLA Int. Workshop on Software Factories. (2005)

[104] Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Software **20**(5) (2003) 42–45

[105] Eclipse Org.: UML2 Project (2007) `http://www.eclipse.org/modeling/mdt/?project=uml2tools#uml2tools`.

[106] Vukmirovic, O., Tilghman, S.: Exploring genome space. Nature **405**(6788) (2000) 820–2

[107] Täubner, C., Mathiak, B., Kupfer, A., Fleischer, N., Eckstein, S.: Modelling and simulation of the tlr4 pathway with coloured petri nets. In: EMBS'06. 28th Annual International Conference of the IEEE. (2006) 2009–2012

[108] Täubner, C., Merker, T.: Discrete modelling of the ethylene-pathway. In: ICDE Workshops. (2005) 1152

[109] Fenton, N.E., Neil, M.: Software metrics: roadmap. In: ICSE - Future of SE Track. (2000) 357–370

[110] Bernstein, P.A., Halevy, A.Y., Pottinger, R.A.: A vision for management of complex models. SIGMOD Record (ACM Special Interest Group on Management of Data) **29**(4) (2000) 55–63

[111] Melnik, S., Rahm, E., Bernstein, P.A.: Rondo: a programming platform for generic model management. In: SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM Press (2003) 193–204

[112] Song, G., Zhang, K., Kong, J.: Model management through graph transformation. In: VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04), Washington, DC, USA, IEEE Computer Society (2004) 75–82

[113] Kruchten, P.: The Rational Unified Process: an introduction. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)

[114] Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Prentice Hall PTR, Upper Saddle River, NJ, USA (2001)

[115] Batini, C., Lenzerini, M., Navathe, S.B.: A comparative analysis of methodologies for database schema integration. ACM Comput. Surv. **18**(4) (1986) 323–364

[116] Meseguer, J.: Rewriting logic as a semantic framework for concurrency: a progress report. In: Proc. CONCUR'96, Pisa, August 1996, Springer LNCS 1119 (1996) 331–372

[117] Eclipse Organization: The graphical modeling framework (2006) `http://www.eclipse.org/gmf/`.

[118] Varró, D.: Model transformation by example. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: MoDELS. Volume 4199 of Lecture Notes in Computer Science., Springer (2006) 410–424

[119] Bardohl, R., Ehrig, H., de Lara, J., Taentzer, G.: Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. [146] 214–228

[120] Bottoni, P., Taentzer, G., Schürr, A.: Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In: VL '00: Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00), Washington, DC, USA, IEEE Computer Society (2000) 59

[121] Engels, G., Heckel, R., Küster, J.M.: Rule-based specification of behavioral consistency based on the uml meta-model. In: UML '01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, London, UK, Springer-Verlag (2001) 272–286

[122] Varró, D., Varró, G., Pataricza, A.: Designing the automatic transformation of visual languages. Sci. Comput. Program. **44**(2) (2002) 205–227

[123] Kuske, S.: A formal semantics of uml state machines based on structured graph transformation. [145] 241–256

[124] Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In: ICSE, ACM (2002) 105–115

[125] Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In Corradini, A., Ehrig, H., Kreowski, H.J., Rozenberg, G., eds.: ICGT. Volume 2505 of Lecture Notes in Computer Science., Springer (2002) 161–176

[126] de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed graph transformation with node type inheritance. Theor. Comput. Sci. **376**(3) (2007) 139–163

[127] Taentzer, G., Rensink, A.: Ensuring structural constraints in graph-based models with type inheritance. [148] 64–79

[128] Balogh, A., Varró, D.: Advanced model transformation language constructs in the viatra2 framework. [144] 1280–1287

[129] Heckel, R.: Introductory tutorial on foundations and applications of graph transformation. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: ICGT. Volume 4178 of Lecture Notes in Computer Science., Springer (2006) 461–462

[130] Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: An algebraic approach. In: FOCS, IEEE (1973) 167–180

[131] Towards a Basic Theory to Model Model Driven Engineering. In: Workshop on Software Model Engineering, WISME 2004, joint event with UML2004,Lisboa, Portugal, October 11, 2004. (2004)

[132] Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. In Baader, F., ed.: CADE. Volume 2741 of Lecture Notes in Computer Science., Springer (2003) 2–16

[133] Modelware Consortium: The modelware project (2006) `http://www.modelware-ist.org/`.

[134] Favre, J.M.: Cacophony: Metamodel-driven architecture recovery. In: WCRE, IEEE Computer Society (2004) 204–213

[135] Klint, P., Lämmel, R., Verhoef, C.: Towards an engineering discipline for grammarware. ACM Transactions on Software Engineering Methodology **14**(3) (2005) 331–380

[136] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4) (2005) 316–344

[137] Alanen, M., Porres, I.: A relation between context-free grammars and meta object facility metamodels. Technical Report 606, TUCS - Turku Centre for Computer Science, Turku, Finland (2004)

[138] Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: GPCE'06: Proceedings of the fifth international conference on Generative programming and Component Engineering. (2006) To appear.

[139] Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schnekenburger, R., Gérard, S., Jézéquel, J.M.: Model-driven analysis and synthesis of concrete syntax. In: ACM/IEEE 9th International Conference on Model-Driven Engineering Languages and Systems (MODELS'06). (2006) To appear.

[140] Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In Bruel, J.M., ed.: MoDELS Satellite Events. Volume 3844 of Lecture Notes in Computer Science., Springer (2005) 159–168

[141] Kunert, A.: Semi-automatic generation of metamodels and models from grammars and programs. In: Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT). ETAPS. (2006)

[142] Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Initial algebra semantics and continuous algebras. J. ACM **24**(1) (1977) 68–95

[143] Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2007), September 11-14, 2005, Zaragoza, Spain. In: JISBD. (2007)

[144] Haddad, H., ed.: Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006. In Haddad, H., ed.: SAC, ACM (2006)

[145] Gogolla, M., Kobryn, C., eds.: UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings. In Gogolla, M., Kobryn, C., eds.: UML. Volume 2185 of Lecture Notes in Computer Science., Springer (2001)

[146] Wermelinger, M., Margaria, T., eds.: Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004 Barcelona, Spain, March 29 - april 2, 2004, Proceedings. In Wermelinger, M., Margaria, T., eds.: FASE. Volume 2984 of Lecture Notes in Computer Science., Springer (2004)

[147] Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G., eds.: Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings. In Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G., eds.: ICGT. Volume 3256 of Lecture Notes in Computer Science., Springer (2004)

[148] Cerioli, M., ed.: Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. In Cerioli, M., ed.: FASE. Volume 3442 of Lecture Notes in Computer Science., Springer (2005)

Appendixes

Appendix A

The MOF Theory

```
mod mod#ecore is
  including  OCL-DATATYPE-COLLECTIONS .
  protecting BOOL .
  sorts Cid#ecore EObject#ecore Oid#ecore Property#ecore
    PropertySet#ecore ecore/EAnnotation ecore/EAttribute ecore/EClass
    ecore/EClassifier ecore/EDataType ecore/EEnum ecore/EEnumLiteral
    ecore/EFactory ecore/EModelElement ecore/ENamedElement ecore/EObject
    ecore/EOperation ecore/EPackage ecore/EParameter ecore/EReference
    ecore/EStringToStringMapEntry ecore/EStructuralFeature ecore/ETypedElement
    oid#ecore/EAnnotation oid#ecore/EAttribute oid#ecore/EClass
    oid#ecore/EClassifier oid#ecore/EDataType oid#ecore/EEnum
    oid#ecore/EEnumLiteral oid#ecore/EFactory oid#ecore/EModelElement
    oid#ecore/ENamedElement oid#ecore/EObject oid#ecore/EOperation
    oid#ecore/EPackage oid#ecore/EParameter oid#ecore/EReference
    oid#ecore/EStringToStringMapEntry oid#ecore/EStructuralFeature
    oid#ecore/ETypedElement .
  subsorts Cid#ecore < ecore/EObject .
  subsorts Oid#ecore < oid#ecore/EObject .
  subsorts Property#ecore < PropertySet#ecore .
  subsorts ecore/EAnnotation < ecore/EModelElement .
  subsorts ecore/EAttribute < ecore/EStructuralFeature .
  subsorts ecore/EClass < ecore/EClassifier .
  subsorts ecore/EClassifier < ecore/ENamedElement .
  subsorts ecore/EDataType < ecore/EClassifier .
  subsorts ecore/EEnum < ecore/EDataType .
  subsorts ecore/EEnumLiteral < ecore/ENamedElement .
  subsorts ecore/EFactory < ecore/EModelElement .
  subsorts ecore/EModelElement < Cid#ecore .
  subsorts ecore/ENamedElement < ecore/EModelElement .
  subsorts ecore/EOperation < ecore/ETypedElement .
  subsorts ecore/EPackage < ecore/ENamedElement .
  subsorts ecore/EParameter < ecore/ETypedElement .
  subsorts ecore/EReference < ecore/EStructuralFeature .
  subsorts ecore/EStringToStringMapEntry < Cid#ecore .
  subsorts ecore/EStructuralFeature < ecore/ETypedElement .
  subsorts ecore/ETypedElement < ecore/ENamedElement .
  subsorts oid#ecore/EAnnotation < oid#ecore/EModelElement .
  subsorts oid#ecore/EAttribute < oid#ecore/EStructuralFeature .
  subsorts oid#ecore/EClass < oid#ecore/EClassifier .
  subsorts oid#ecore/EClassifier < oid#ecore/ENamedElement .
  subsorts oid#ecore/EDataType < oid#ecore/EClassifier .
  subsorts oid#ecore/EEnum < oid#ecore/EDataType .
  subsorts oid#ecore/EEnumLiteral < oid#ecore/ENamedElement .
  subsorts oid#ecore/EFactory < oid#ecore/EModelElement .
  subsorts oid#ecore/EModelElement < Oid#ecore .
  subsorts oid#ecore/ENamedElement < oid#ecore/EModelElement .
  subsorts oid#ecore/EObject < AbstractOid .
  subsorts oid#ecore/EOperation < oid#ecore/ETypedElement .
  subsorts oid#ecore/EPackage < oid#ecore/ENamedElement .
  subsorts oid#ecore/EParameter < oid#ecore/ETypedElement .
  subsorts oid#ecore/EReference < oid#ecore/EStructuralFeature .
  subsorts oid#ecore/EStringToStringMapEntry < Oid#ecore .
  subsorts oid#ecore/EStructuralFeature < oid#ecore/ETypedElement .
  subsorts oid#ecore/ETypedElement < oid#ecore/ENamedElement .


  --- this constant is needed to apply downTerm to a constant that represents the name of a class
  op ecore/EObject : -> ecore/EObject [ctor] .
  op nullOid#ecore : -> [Oid#ecore] .
  op class : EObject#ecore -> Cid#ecore .
  op getPropertySet : EObject#ecore -> PropertySet#ecore .
  op noneProperty#ecore : -> PropertySet#ecore .
  op nullEObject#ecore : -> [EObject#ecore] .
  op oid : EObject#ecore -> Oid#ecore .
  op <_:_|_> : Oid#ecore Cid#ecore PropertySet#ecore -> EObject#ecore [obj ctor
    format (d n++i ni ni ni ni n--i d)] .
  op _‘,_ : PropertySet#ecore PropertySet#ecore -> PropertySet#ecore [assoc
    comm id: noneProperty#ecore ctor format (d d ni d)] .
  op ecore/EAnnotation : -> ecore/EAnnotation [ctor] .
  op ecore/EAnnotation/contents/0 : -> Property#ecore [ctor] .
  op ecore/EAnnotation/contents‘:_ : OrderedSet{Oid} -> Property#ecore [ctor] .
  op ecore/EAnnotation/details/0 : -> Property#ecore [ctor] .
  op ecore/EAnnotation/details‘:_ : OrderedSet{Oid} -> Property#ecore [ctor] .
  op ecore/EAnnotation/eModelElement/0 : -> Property#ecore [ctor] .
  op ecore/EAnnotation/eModelElement‘:_ : [Oid#ecore]  -> Property#ecore [ctor] .
  op ecore/EAnnotation/references/0 : -> Property#ecore [ctor] .
  op ecore/EAnnotation/references‘:_ : OrderedSet{Oid} -> Property#ecore [ctor]
    .
  op ecore/EAnnotation/source/0 : -> Property#ecore [ctor] .
  op ecore/EAnnotation/source‘:_ : String -> Property#ecore [ctor] .
  op ecore/EAttribute : -> ecore/EAttribute [ctor] .
  op ecore/EAttribute/eAttributeType/0 : -> Property#ecore [ctor] .
  op ecore/EAttribute/eAttributeType‘:_ : Oid#ecore -> Property#ecore [ctor] .
  op ecore/EAttribute/iD/0 : -> Property#ecore [ctor] .
  op ecore/EAttribute/iD‘:_ : Bool -> Property#ecore [ctor] .
  op ecore/EClass : -> ecore/EClass [ctor] .
  op ecore/EClass/abstract/0 : -> Property#ecore [ctor] .
  op ecore/EClass/abstract‘:_ : Bool -> Property#ecore [ctor] .
  op ecore/EClass/eAllAttributes/0 : -> Property#ecore [ctor] .
  op ecore/EClass/eAllAttributes‘:_ : OrderedSet{Oid} -> Property#ecore [ctor]
    .
  op ecore/EClass/eAllContainments/0 : -> Property#ecore [ctor] .
  op ecore/EClass/eAllContainments‘:_ : OrderedSet{Oid} -> Property#ecore [
    ctor] .
  op ecore/EClass/eAllOperations/0 : -> Property#ecore [ctor] .
  op ecore/EClass/eAllOperations‘:_ : OrderedSet{Oid} -> Property#ecore [ctor]
    .
  op ecore/EClass/eAllReferences/0 : -> Property#ecore [ctor] .
  op ecore/EClass/eAllReferences‘:_ : OrderedSet{Oid} -> Property#ecore [ctor]
    .
  op ecore/EClass/eAllStructuralFeatures/0 : -> Property#ecore [ctor] .
  op ecore/EClass/eAllStructuralFeatures‘:_ : OrderedSet{Oid} -> Property#ecore
    [ctor] .
  op ecore/EClass/eAllSuperTypes/0 : -> Property#ecore [ctor] .
  op ecore/EClass/eAllSuperTypes‘:_ : OrderedSet{Oid} -> Property#ecore [ctor]
    .
  op ecore/EClass/eAttributes/0 : -> Property#ecore [ctor] .
  op ecore/EClass/eAttributes‘:_ : OrderedSet{Oid} -> Property#ecore [ctor] .
  op ecore/EClass/eIDAttribute/0 : -> Property#ecore [ctor] .
  op ecore/EClass/eIDAttribute‘:_ : [Oid#ecore]  -> Property#ecore [ctor] .
  op ecore/EClass/eOperations/0 : -> Property#ecore [ctor] .
  op ecore/EClass/eOperations‘:_ : OrderedSet{Oid} -> Property#ecore [ctor] .
  op ecore/EClass/eReferences/0 : -> Property#ecore [ctor] .
  op ecore/EClass/eReferences‘:_ : OrderedSet{Oid} -> Property#ecore [ctor] .
  op ecore/EClass/eStructuralFeatures/0 : -> Property#ecore [ctor] .
  op ecore/EClass/eStructuralFeatures‘:_ : OrderedSet{Oid} -> Property#ecore [
    ctor] .
  op ecore/EClass/eSuperTypes/0 : -> Property#ecore [ctor] .
  op ecore/EClass/eSuperTypes‘:_ : OrderedSet{Oid} -> Property#ecore [ctor] .
  op ecore/EClass/interface/0 : -> Property#ecore [ctor] .
  op ecore/EClass/interface‘:_ : Bool -> Property#ecore [ctor] .
  op ecore/EClassifier : -> ecore/EClassifier [ctor] .
  op ecore/EClassifier/defaultValue/0 : -> Property#ecore [ctor] .
  op ecore/EClassifier/defaultValue‘:_ : String -> Property#ecore [ctor] .
  op ecore/EClassifier/ePackage/0 : -> Property#ecore [ctor] .
  op ecore/EClassifier/ePackage‘:_ : [Oid#ecore]  -> Property#ecore [ctor] .
  op ecore/EClassifier/instanceClass/0 : -> Property#ecore [ctor] .
  op ecore/EClassifier/instanceClassName/0 : -> Property#ecore [ctor] .
  op ecore/EClassifier/instanceClassName‘:_ : String -> Property#ecore [ctor] .
  op ecore/EClassifier/instanceClass‘:_ : String -> Property#ecore [ctor] .
  op ecore/EDataType : -> ecore/EDataType [ctor] .
  op ecore/EDataType/serializable/0 : -> Property#ecore [ctor] .
  op ecore/EDataType/serializable‘:_ : Bool -> Property#ecore [ctor] .
  op ecore/EEnum : -> ecore/EEnum [ctor] .
  op ecore/EEnum/eLiterals/0 : -> Property#ecore [ctor] .
  op ecore/EEnum/eLiterals‘:_ : OrderedSet{Oid} -> Property#ecore [ctor] .
  op ecore/EEnumLiteral : -> ecore/EEnumLiteral [ctor] .
  op ecore/EEnumLiteral/eEnum/0 : -> Property#ecore [ctor] .
  op ecore/EEnumLiteral/eEnum‘:_ : [Oid#ecore]  -> Property#ecore [ctor] .
  op ecore/EEnumLiteral/instance/0 : -> Property#ecore [ctor] .
  op ecore/EEnumLiteral/instance`:_ : String -> Property#ecore [ctor] .
  op ecore/EEnumLiteral/literal/0 : -> Property#ecore [ctor] .
  op ecore/EEnumLiteral/literal`:_ : String -> Property#ecore [ctor] .
  op ecore/EEnumLiteral/value/0 : -> Property#ecore [ctor] .
  op ecore/EEnumLiteral/value`:_ : Int -> Property#ecore [ctor] .
  op ecore/EFactory : -> ecore/EFactory [ctor] .
  op ecore/EFactory/ePackage/0 : -> Property#ecore [ctor] .
  op ecore/EFactory/ePackage`:_ : Oid#ecore -> Property#ecore [ctor] .
  op ecore/EModelElement : -> ecore/EModelElement [ctor] .
  op ecore/EModelElement/eAnnotations/0 : -> Property#ecore [ctor] .
  op ecore/EModelElement/eAnnotations`:_ : OrderedSet{Oid} -> Property#ecore [
    ctor] .
  op ecore/ENamedElement : -> ecore/ENamedElement [ctor] .
  op ecore/ENamedElement/name/0 : -> Property#ecore [ctor] .
  op ecore/ENamedElement/name`:_ : String -> Property#ecore [ctor] .
  op ecore/EOperation : -> ecore/EOperation [ctor] .
  op ecore/EOperation/eContainingClass/0 : -> Property#ecore [ctor] .
  op ecore/EOperation/eContainingClass`:_ : [Oid#ecore]  -> Property#ecore [ctor] .
  op ecore/EOperation/eExceptions/0 : -> Property#ecore [ctor] .
  op ecore/EOperation/eExceptions`:_ : OrderedSet{Oid} -> Property#ecore [ctor]
    .
  op ecore/EOperation/eParameters/0 : -> Property#ecore [ctor] .
  op ecore/EOperation/eParameters`:_ : OrderedSet{Oid} -> Property#ecore [ctor]
    .
  op ecore/EPackage : -> ecore/EPackage [ctor] .
  op ecore/EPackage/eClassifiers/0 : -> Property#ecore [ctor] .
  op ecore/EPackage/eClassifiers`:_ : OrderedSet{Oid} -> Property#ecore [ctor]
    .
  op ecore/EPackage/eFactoryInstance/0 : -> Property#ecore [ctor] .
  op ecore/EPackage/eFactoryInstance`:_ : Oid#ecore -> Property#ecore [ctor] .
  op ecore/EPackage/eSubpackages/0 : -> Property#ecore [ctor] .
  op ecore/EPackage/eSubpackages`:_ : OrderedSet{Oid} -> Property#ecore [ctor]
    .
  op ecore/EPackage/eSuperPackage/0 : -> Property#ecore [ctor] .
  op ecore/EPackage/eSuperPackage`:_ : [Oid#ecore]  -> Property#ecore [ctor] .
  op ecore/EPackage/nsPrefix/0 : -> Property#ecore [ctor] .
  op ecore/EPackage/nsPrefix`:_ : String -> Property#ecore [ctor] .
  op ecore/EPackage/nsURI/0 : -> Property#ecore [ctor] .
  op ecore/EPackage/nsURI`:_ : String -> Property#ecore [ctor] .
  op ecore/EParameter : -> ecore/EParameter [ctor] .
  op ecore/EParameter/eOperation/0 : -> Property#ecore [ctor] .
  op ecore/EParameter/eOperation`:_ : [Oid#ecore]  -> Property#ecore [ctor] .
  op ecore/EReference : -> ecore/EReference [ctor] .
  op ecore/EReference/container/0 : -> Property#ecore [ctor] .
  op ecore/EReference/container`:_ : Bool -> Property#ecore [ctor] .
  op ecore/EReference/containment/0 : -> Property#ecore [ctor] .
  op ecore/EReference/containment`:_ : Bool -> Property#ecore [ctor] .
  op ecore/EReference/eOpposite/0 : -> Property#ecore [ctor] .
  op ecore/EReference/eOpposite`:_ : [Oid#ecore]  -> Property#ecore [ctor] .
  op ecore/EReference/eReferenceType/0 : -> Property#ecore [ctor] .
  op ecore/EReference/eReferenceType`:_ : Oid#ecore -> Property#ecore [ctor] .
  op ecore/EReference/resolveProxies/0 : -> Property#ecore [ctor] .
  op ecore/EReference/resolveProxies`:_ : Bool -> Property#ecore [ctor] .
  op ecore/EStringToStringMapEntry : -> ecore/EStringToStringMapEntry [ctor] .
  op ecore/EStringToStringMapEntry/key/0 : -> Property#ecore [ctor] .
  op ecore/EStringToStringMapEntry/key`:_ : String -> Property#ecore [ctor] .
  op ecore/EStringToStringMapEntry/value/0 : -> Property#ecore [ctor] .
  op ecore/EStringToStringMapEntry/value`:_ : String -> Property#ecore [ctor] .
  op ecore/EStructuralFeature : -> ecore/EStructuralFeature [ctor] .
  op ecore/EStructuralFeature/changeable/0 : -> Property#ecore [ctor] .
  op ecore/EStructuralFeature/changeable`:_ : Bool -> Property#ecore [ctor] .
  op ecore/EStructuralFeature/defaultValue/0 : -> Property#ecore [ctor] .
  op ecore/EStructuralFeature/defaultValueLiteral/0 : -> Property#ecore [ctor]
    .
  op ecore/EStructuralFeature/defaultValueLiteral`:_ : String -> Property#ecore
    [ctor] .
  op ecore/EStructuralFeature/defaultValue`:_ : String -> Property#ecore [ctor]
    .
  op ecore/EStructuralFeature/derived/0 : -> Property#ecore [ctor] .
  op ecore/EStructuralFeature/derived`:_ : Bool -> Property#ecore [ctor] .
  op ecore/EStructuralFeature/eContainingClass/0 : -> Property#ecore [ctor] .
  op ecore/EStructuralFeature/eContainingClass`:_ : [Oid#ecore]  -> Property#ecore [
    ctor] .
  op ecore/EStructuralFeature/transient/0 : -> Property#ecore [ctor] .
  op ecore/EStructuralFeature/transient`:_ : Bool -> Property#ecore [ctor] .
  op ecore/EStructuralFeature/unsettable/0 : -> Property#ecore [ctor] .
  op ecore/EStructuralFeature/unsettable`:_ : Bool -> Property#ecore [ctor] .
  op ecore/EStructuralFeature/volatile/0 : -> Property#ecore [ctor] .
  op ecore/EStructuralFeature/volatile`:_ : Bool -> Property#ecore [ctor] .
  op ecore/ETypedElement : -> ecore/ETypedElement [ctor] .
```

```
  op ecore/ETypedElement/eType/0 : -> Property#ecore [ctor] .
  op ecore/ETypedElement/eType`:_ : [Oid#ecore]  -> Property#ecore [ctor] .
  op ecore/ETypedElement/lowerBound/0 : -> Property#ecore [ctor] .
  op ecore/ETypedElement/lowerBound`:_ : Int -> Property#ecore [ctor] .
  op ecore/ETypedElement/many/0 : -> Property#ecore [ctor] .
  op ecore/ETypedElement/many`:_ : Bool -> Property#ecore [ctor] .
  op ecore/ETypedElement/ordered/0 : -> Property#ecore [ctor] .
  op ecore/ETypedElement/ordered`:_ : Bool -> Property#ecore [ctor] .
  op ecore/ETypedElement/required/0 : -> Property#ecore [ctor] .
  op ecore/ETypedElement/required`:_ : Bool -> Property#ecore [ctor] .
  op ecore/ETypedElement/unique/0 : -> Property#ecore [ctor] .
  op ecore/ETypedElement/unique`:_ : Bool -> Property#ecore [ctor] .
  op ecore/ETypedElement/upperBound/0 : -> Property#ecore [ctor] .
  op ecore/ETypedElement/upperBound`:_ : Int -> Property#ecore [ctor] .
  op oid#ecore/EAnnotation : Qid -> oid#ecore/EAnnotation [ctor] .
  op oid#ecore/EAttribute : Qid -> oid#ecore/EAttribute [ctor] .
  op oid#ecore/EClass : Qid -> oid#ecore/EClass [ctor] .
  op oid#ecore/EClassifier : Qid -> oid#ecore/EClassifier [ctor] .
  op oid#ecore/EDataType : Qid -> oid#ecore/EDataType [ctor] .
  op oid#ecore/EEnum : Qid -> oid#ecore/EEnum [ctor] .
  op oid#ecore/EEnumLiteral : Qid -> oid#ecore/EEnumLiteral [ctor] .
  op oid#ecore/EFactory : Qid -> oid#ecore/EFactory [ctor] .
  op oid#ecore/EModelElement : Qid -> oid#ecore/EModelElement [ctor] .
  op oid#ecore/ENamedElement : Qid -> oid#ecore/ENamedElement [ctor] .
  op oid#ecore/EObject : Qid -> oid#ecore/EObject [ctor] .
  op oid#ecore/EOperation : Qid -> oid#ecore/EOperation [ctor] .
  op oid#ecore/EPackage : Qid -> oid#ecore/EPackage [ctor] .
  op oid#ecore/EParameter : Qid -> oid#ecore/EParameter [ctor] .
  op oid#ecore/EReference : Qid -> oid#ecore/EReference [ctor] .
  op oid#ecore/EStringToStringMapEntry : Qid ->
    oid#ecore/EStringToStringMapEntry [ctor] .
  op oid#ecore/EStructuralFeature : Qid -> oid#ecore/EStructuralFeature [ctor]
    .
  op oid#ecore/ETypedElement : Qid -> oid#ecore/ETypedElement [ctor] .
  eq class (< Oid:Oid#ecore : CID:Cid#ecore | PS:PropertySet#ecore >) =
    CID:Cid#ecore .
  eq getPropertySet (< Oid:Oid#ecore : CID:Cid#ecore | PS:PropertySet#ecore >)
    = PS:PropertySet#ecore .
  eq oid (< Oid:Oid#ecore : CID:Cid#ecore | PS:PropertySet#ecore >) =
    Oid:Oid#ecore .
endm


view ecore from TH-EOBJECT to mod#ecore is
  sort EObject to EObject#ecore .
  sort Oid to Oid#ecore .
  sort Cid to Cid#ecore .
  sort Property to Property#ecore .
  sort PropertySet to PropertySet#ecore .
  op nullOid to nullOid#ecore .
  op noneProperty to noneProperty#ecore .
  op nullEObject to nullEObject#ecore .
endv
```

Appendix B

The RDBMS Metamodel Definition

```
<< <
  oid#ecore/EAttribute(
    '#//Column/nnv)
  :
  ecore/EAttribute
  |
  ecore/EAttribute/iD,
  ecore/EModelElement/eAnnotations,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/transient,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/many,
  ecore/EAttribute/eAttributeType : oid#ecore/EDataType(
    'http://www.eclipse.org/emf/2002/Ecore#//EBoolean),
  ecore/ENamedElement/name : "nnv",
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "false",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//Column),
  ecore/ETypedElement/eType : oid#ecore/EDataType(
    'http://www.eclipse.org/emf/2002/Ecore#//EBoolean),
  ecore/ETypedElement/lowerBound : 1,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/required : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : 1
>
<
  oid#ecore/EAttribute(
    '#//Column/type)
  :
  ecore/EAttribute
  |
  ecore/EAttribute/iD,
  ecore/EModelElement/eAnnotations,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/transient,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/many,
  ecore/EAttribute/eAttributeType : oid#ecore/EEnum(
    '#//RDataType),
  ecore/ENamedElement/name : "type",
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "VARCHAR",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//Column),
  ecore/ETypedElement/eType : oid#ecore/EEnum(
    '#//RDataType),
  ecore/ETypedElement/lowerBound : 1,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/required : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : 1
>
<
  oid#ecore/EAttribute(
    '#//RModelEment/name)
  :
  ecore/EAttribute
  |
  ecore/EAttribute/iD,
  ecore/EModelElement/eAnnotations,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/transient,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/lowerBound,
  ecore/ETypedElement/many,
  ecore/ETypedElement/required,
  ecore/EAttribute/eAttributeType : oid#ecore/EDataType(
    'http://www.eclipse.org/emf/2002/Ecore#//EString),
  ecore/ENamedElement/name : "name",
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//RModelEment),
  ecore/ETypedElement/eType : oid#ecore/EDataType(
    'http://www.eclipse.org/emf/2002/Ecore#//EString),
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : 1
>
<
  oid#ecore/EClass('#//Column)
  :
  ecore/EClass
  |
  ecore/EClass/abstract,
  ecore/EClass/eAllContainments,
  ecore/EClass/eAllOperations,
  ecore/EClass/eIDAttribute,
  ecore/EClass/eOperations,
  ecore/EClass/interface,
  ecore/EClassifier/instanceClassName,
  ecore/EModelElement/eAnnotations,
  ecore/EClass/eAllAttributes : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name) ::
    oid#ecore/EAttribute(
    '#//Column/nnv) ::
    oid#ecore/EAttribute(
    '#//Column/type)},
  ecore/EClass/eAllReferences : OrderedSet{oid#ecore/EReference(
    '#//Column/foreignKey) ::
    oid#ecore/EReference(
    '#//Column/key) ::
    oid#ecore/EReference(
    '#//Column/owner)},
  ecore/EClass/eAllStructuralFeatures : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name) ::
    oid#ecore/EAttribute(
    '#//Column/nnv) ::
    oid#ecore/EAttribute(
    '#//Column/type) ::
    oid#ecore/EReference(
    '#//Column/foreignKey) ::
    oid#ecore/EReference(
    '#//Column/key) ::
    oid#ecore/EReference(
    '#//Column/owner)},
  ecore/EClass/eAllSuperTypes : OrderedSet{oid#ecore/EClass(
    '#//RModelEment)},
  ecore/EClass/eAttributes : OrderedSet{oid#ecore/EAttribute(
    '#//Column/nnv) ::
    oid#ecore/EAttribute(
    '#//Column/type)},
  ecore/EClass/eReferences : OrderedSet{oid#ecore/EReference(
    '#//Column/foreignKey) ::
    oid#ecore/EReference(
    '#//Column/key) ::
    oid#ecore/EReference(
    '#//Column/owner)},
  ecore/EClass/eStructuralFeatures : OrderedSet{oid#ecore/EAttribute(
    '#//Column/nnv) ::
    oid#ecore/EAttribute(
    '#//Column/type) ::
    oid#ecore/EReference(
    '#//Column/foreignKey) ::
    oid#ecore/EReference(
    '#//Column/key) ::
    oid#ecore/EReference(
    '#//Column/owner)},
  ecore/EClass/eSuperTypes : OrderedSet{oid#ecore/EClass(
    '#//RModelEment)},
  ecore/EClassifier/defaultValue : "0",
  ecore/EClassifier/ePackage : oid#ecore/EPackage(
    '#/),
  ecore/EClassifier/instanceClass : "0",
  ecore/ENamedElement/name : "Column"
>
<
  oid#ecore/EClass('#//ForeignKey)
  :
  ecore/EClass
  |
  ecore/EClass/abstract,
  ecore/EClass/eAllContainments,
  ecore/EClass/eAllOperations,
  ecore/EClass/eAttributes,
  ecore/EClass/eIDAttribute,
  ecore/EClass/eOperations,
  ecore/EClass/interface,
  ecore/EClassifier/instanceClassName,
  ecore/EModelElement/eAnnotations,
  ecore/EClass/eAllAttributes : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name)},
  ecore/EClass/eAllReferences : OrderedSet{oid#ecore/EReference(
    '#//ForeignKey/refersTo) ::
    oid#ecore/EReference(
    '#//ForeignKey/column) ::
    oid#ecore/EReference(
    '#//ForeignKey/owner)},
  ecore/EClass/eAllStructuralFeatures : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name) ::
    oid#ecore/EReference(
    '#//ForeignKey/refersTo) ::
    oid#ecore/EReference(
    '#//ForeignKey/column) ::
    oid#ecore/EReference(
    '#//ForeignKey/owner)},
  ecore/EClass/eAllSuperTypes : OrderedSet{oid#ecore/EClass(
    '#//RModelEment)},
  ecore/EClass/eReferences : OrderedSet{oid#ecore/EReference(
    '#//ForeignKey/refersTo) ::
    oid#ecore/EReference(
    '#//ForeignKey/column) ::
    oid#ecore/EReference(
    '#//ForeignKey/owner)},
  ecore/EClass/eStructuralFeatures : OrderedSet{oid#ecore/EReference(
    '#//ForeignKey/refersTo) ::
    oid#ecore/EReference(
    '#//ForeignKey/column) ::
    oid#ecore/EReference(
    '#//ForeignKey/owner)},
  ecore/EClass/eSuperTypes : OrderedSet{oid#ecore/EClass(
    '#//RModelEment)},
  ecore/EClassifier/defaultValue : "0",
  ecore/EClassifier/ePackage : oid#ecore/EPackage(
    '#/),
  ecore/EClassifier/instanceClass : "0",
  ecore/ENamedElement/name : "ForeignKey"
>
<
  oid#ecore/EClass('#//Key)
  :
  ecore/EClass
  |
  ecore/EClass/abstract,
  ecore/EClass/eAllContainments,
  ecore/EClass/eAllOperations,
  ecore/EClass/eAttributes,
  ecore/EClass/eIDAttribute,
  ecore/EClass/eOperations,
  ecore/EClass/interface,
  ecore/EClassifier/instanceClassName,
  ecore/EModelElement/eAnnotations,
  ecore/EClass/eAllAttributes : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name)},
  ecore/EClass/eAllReferences : OrderedSet{oid#ecore/EReference(
    '#//Key/owner) ::
    oid#ecore/EReference(
    '#//Key/column)},
  ecore/EClass/eAllStructuralFeatures : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name) ::
    oid#ecore/EReference(
    '#//Key/owner) ::
    oid#ecore/EReference(
    '#//Key/column)},
  ecore/EClass/eAllSuperTypes : OrderedSet{oid#ecore/EClass(
    '#//RModelEment)},
  ecore/EClass/eReferences : OrderedSet{oid#ecore/EReference(
    '#//Key/owner) ::
    oid#ecore/EReference(
    '#//Key/column)},
  ecore/EClass/eStructuralFeatures : OrderedSet{oid#ecore/EReference(
    '#//Key/owner) ::
    oid#ecore/EReference(
    '#//Key/column)},
  ecore/EClass/eSuperTypes : OrderedSet{oid#ecore/EClass(
    '#//RModelEment)},
  ecore/EClassifier/defaultValue : "0",
  ecore/EClassifier/ePackage : oid#ecore/EPackage(
    '#/),
  ecore/EClassifier/instanceClass : "0",
  ecore/ENamedElement/name : "Key"
>
<
  oid#ecore/EClass('#//RModelEment)
  :
  ecore/EClass
  |
  ecore/EClass/abstract,
  ecore/EClass/eAllContainments,
  ecore/EClass/eAllOperations,
  ecore/EClass/eAllReferences,
  ecore/EClass/eAllSuperTypes,
  ecore/EClass/eIDAttribute,
  ecore/EClass/eOperations,
  ecore/EClass/eReferences,
  ecore/EClass/eSuperTypes,
  ecore/EClass/interface,
  ecore/EClassifier/instanceClassName,
  ecore/EModelElement/eAnnotations,
  ecore/EClass/eAllAttributes : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name)},
  ecore/EClass/eAllStructuralFeatures : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name)},
  ecore/EClass/eAttributes : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name)},
  ecore/EClass/eStructuralFeatures : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name)},
  ecore/EClassifier/defaultValue : "0",
  ecore/EClassifier/ePackage : oid#ecore/EPackage(
    '#/),
  ecore/EClassifier/instanceClass : "0",
  ecore/ENamedElement/name : "RModelEment"
>
<
  oid#ecore/EClass('#//Schema)
  :
  ecore/EClass
  |
  ecore/EClass/abstract,
  ecore/EClass/eAllOperations,
  ecore/EClass/eAttributes,
  ecore/EClass/eIDAttribute,
  ecore/EClass/eOperations,
  ecore/EClass/interface,
  ecore/EClassifier/instanceClassName,
  ecore/EModelElement/eAnnotations,
  ecore/EClass/eAllAttributes : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name)},
  ecore/EClass/eAllContainments : OrderedSet{oid#ecore/EReference(
    '#//Schema/tables)},
  ecore/EClass/eAllReferences : OrderedSet{oid#ecore/EReference(
    '#//Schema/tables)},
  ecore/EClass/eAllStructuralFeatures : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name) ::
    oid#ecore/EReference(
    '#//Schema/tables)},
  ecore/EClass/eAllSuperTypes : OrderedSet{oid#ecore/EClass(
    '#//RModelEment)},
  ecore/EClass/eReferences : OrderedSet{oid#ecore/EReference(
    '#//Schema/tables)},
  ecore/EClass/eStructuralFeatures : OrderedSet{oid#ecore/EReference(
    '#//Schema/tables)},
  ecore/EClass/eSuperTypes : OrderedSet{oid#ecore/EClass(
    '#//RModelEment)},
  ecore/EClassifier/defaultValue : "0",
  ecore/EClassifier/ePackage : oid#ecore/EPackage(
    '#/),
  ecore/EClassifier/instanceClass : "0",
  ecore/ENamedElement/name : "Schema"
>
<
  oid#ecore/EClass('#//Table)
  :
  ecore/EClass
  |
  ecore/EClass/abstract,
  ecore/EClass/eAllOperations,
  ecore/EClass/eAttributes,
  ecore/EClass/eIDAttribute,
  ecore/EClass/eOperations,
  ecore/EClass/interface,
  ecore/EClassifier/instanceClassName,
  ecore/EModelElement/eAnnotations,
  ecore/EClass/eAllAttributes : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name)},
  ecore/EClass/eAllContainments : OrderedSet{oid#ecore/EReference(
    '#//Table/key) ::
    oid#ecore/EReference(
    '#//Table/foreignKey) ::
    oid#ecore/EReference(
    '#//Table/column)},
  ecore/EClass/eAllReferences : OrderedSet{oid#ecore/EReference(
    '#//Table/key) ::
    oid#ecore/EReference(
    '#//Table/foreignKey) ::
    oid#ecore/EReference(
    '#//Table/column) ::
    oid#ecore/EReference(
    '#//Table/schema)},
  ecore/EClass/eAllStructuralFeatures : OrderedSet{oid#ecore/EAttribute(
    '#//RModelEment/name) ::
    oid#ecore/EReference(
    '#//Table/key) ::
    oid#ecore/EReference(
    '#//Table/foreignKey) ::
    oid#ecore/EReference(
    '#//Table/column) ::
    oid#ecore/EReference(
    '#//Table/schema)},
  ecore/EClass/eAllSuperTypes : OrderedSet{oid#ecore/EClass(
    '#//RModelEment)},
  ecore/EClass/eReferences : OrderedSet{oid#ecore/EReference(
    '#//Table/key) ::
    oid#ecore/EReference(
    '#//Table/foreignKey) ::
    oid#ecore/EReference(
    '#//Table/column) ::
    oid#ecore/EReference(
    '#//Table/schema)},
  ecore/EClass/eStructuralFeatures : OrderedSet{oid#ecore/EReference(
    '#//Table/key) ::
    oid#ecore/EReference(
    '#//Table/foreignKey) ::
    oid#ecore/EReference(
    '#//Table/column) ::
    oid#ecore/EReference(
    '#//Table/schema)},
  ecore/EClass/eSuperTypes : OrderedSet{oid#ecore/EClass(
    '#//RModelEment)},
  ecore/EClassifier/defaultValue : "0",
  ecore/EClassifier/ePackage : oid#ecore/EPackage(
    '#/),
  ecore/EClassifier/instanceClass : "0",
  ecore/ENamedElement/name : "Table"
>
<
  oid#ecore/EEnum('#//RDataType)
  :
  ecore/EEnum
  |
  ecore/EClassifier/instanceClassName,
  ecore/EModelElement/eAnnotations,
  ecore/EClassifier/defaultValue : "VARCHAR",
  ecore/EClassifier/ePackage : oid#ecore/EPackage(
    '#/),
  ecore/EClassifier/instanceClass : "0",
  ecore/EDataType/serializable : true,
  ecore/EEnum/eLiterals : OrderedSet{oid#ecore/EEnumLiteral(
    '#//RDataType/VARCHAR) ::
    oid#ecore/EEnumLiteral(
    '#//RDataType/NUMBER) ::
    oid#ecore/EEnumLiteral(
    '#//RDataType/BOOLEAN) ::
    oid#ecore/EEnumLiteral(
    '#//RDataType/DATE) ::
    oid#ecore/EEnumLiteral(
    '#//RDataType/DECIMAL)},
  ecore/ENamedElement/name : "RDataType"
>
<
  oid#ecore/EEnumLiteral(
    '#//RDataType/BOOLEAN)
  :
  ecore/EEnumLiteral
  |
  ecore/EEnumLiteral/value,
  ecore/EModelElement/eAnnotations,
  ecore/EEnumLiteral/eEnum : oid#ecore/EEnum(
    '#//RDataType),
  ecore/EEnumLiteral/instance : "BOOLEAN",
  ecore/EEnumLiteral/literal : "BOOLEAN",
  ecore/ENamedElement/name : "BOOLEAN"
>
<
  oid#ecore/EEnumLiteral(
    '#//RDataType/DATE)
  :
  ecore/EEnumLiteral
  |
  ecore/EEnumLiteral/value,
  ecore/EModelElement/eAnnotations,
  ecore/EEnumLiteral/eEnum : oid#ecore/EEnum(
    '#//RDataType),
  ecore/EEnumLiteral/instance : "DATE",
  ecore/EEnumLiteral/literal : "DATE",
  ecore/ENamedElement/name : "DATE"
>
<
  oid#ecore/EEnumLiteral(
    '#//RDataType/DECIMAL)
  :
  ecore/EEnumLiteral
  |
  ecore/EEnumLiteral/value,
  ecore/EModelElement/eAnnotations,
  ecore/EEnumLiteral/eEnum : oid#ecore/EEnum(
    '#//RDataType),
  ecore/EEnumLiteral/instance : "DECIMAL",
  ecore/EEnumLiteral/literal : "DECIMAL",
  ecore/ENamedElement/name : "DECIMAL"
>
<
  oid#ecore/EEnumLiteral(
    '#//RDataType/NUMBER)
  :
  ecore/EEnumLiteral
  |
  ecore/EEnumLiteral/value,
  ecore/EModelElement/eAnnotations,
  ecore/EEnumLiteral/eEnum : oid#ecore/EEnum(
    '#//RDataType),
  ecore/EEnumLiteral/instance : "NUMBER",
  ecore/EEnumLiteral/literal : "NUMBER",
  ecore/ENamedElement/name : "NUMBER"
>
<
  oid#ecore/EEnumLiteral(
    '#//RDataType/VARCHAR)
  :
  ecore/EEnumLiteral
  |
  ecore/EEnumLiteral/value,
  ecore/EModelElement/eAnnotations,
  ecore/EEnumLiteral/eEnum : oid#ecore/EEnum(
    '#//RDataType),
  ecore/EEnumLiteral/instance : "VARCHAR",
  ecore/EEnumLiteral/literal : "VARCHAR",
  ecore/ENamedElement/name : "VARCHAR"
>
<
  oid#ecore/EPackage('#/)
  :
  ecore/EPackage
  |
  ecore/EModelElement/eAnnotations,
  ecore/EPackage/eFactoryInstance,
  ecore/EPackage/eSubpackages,
  ecore/EPackage/eSuperPackage,
  ecore/ENamedElement/name : "rdbms",
  ecore/EPackage/eClassifiers : OrderedSet{oid#ecore/EClass(
    '#//Schema) ::
    oid#ecore/EClass(
    '#//RModelEment) ::
    oid#ecore/EClass('#//Table) ::
    oid#ecore/EClass('#//Column) ::
    oid#ecore/EClass(
    '#//ForeignKey) ::
    oid#ecore/EClass('#//Key) ::
    oid#ecore/EEnum(
    '#//RDataType)},
  ecore/EPackage/nsPrefix : "rdbms",
  ecore/EPackage/nsURI : "http:///es.upv.dsic.issi/moment/rdbms"
>
<
  oid#ecore/EReference(
    '#//Column/foreignKey)
  :
  ecore/EReference
  |
  ecore/EModelElement/eAnnotations,
  ecore/EReference/container,
  ecore/EReference/containment,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/transient,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/lowerBound,
  ecore/ETypedElement/required,
  ecore/ENamedElement/name : "foreignKey",
  ecore/EReference/eOpposite : oid#ecore/EReference(
    '#//ForeignKey/column),
  ecore/EReference/eReferenceType : oid#ecore/EClass(
    '#//ForeignKey),
  ecore/EReference/resolveProxies : true,
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//Column),
  ecore/ETypedElement/eType : oid#ecore/EClass(
    '#//ForeignKey),
  ecore/ETypedElement/many : true,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : -1
>
<
  oid#ecore/EReference(
    '#//Column/key)
  :
  ecore/EReference
  |
  ecore/EModelElement/eAnnotations,
  ecore/EReference/container,
  ecore/EReference/containment,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/transient,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/lowerBound,
  ecore/ETypedElement/required,
  ecore/ENamedElement/name : "key",
  ecore/EReference/eOpposite : oid#ecore/EReference(
    '#//Key/column),
  ecore/EReference/eReferenceType : oid#ecore/EClass(
    '#//Key),
  ecore/EReference/resolveProxies : true,
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//Column),
  ecore/ETypedElement/eType : oid#ecore/EClass(
    '#//Key),
  ecore/ETypedElement/many : true,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : -1
>
<
  oid#ecore/EReference(
    '#//Column/owner)
  :
  ecore/EReference
  |
  ecore/EModelElement/eAnnotations,
  ecore/EReference/containment,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/many,
  ecore/ENamedElement/name : "owner",
  ecore/EReference/container : true,
  ecore/EReference/eOpposite : oid#ecore/EReference(
    '#//Table/column),
  ecore/EReference/eReferenceType : oid#ecore/EClass(
    '#//Table),
  ecore/EReference/resolveProxies : true,
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//Column),
  ecore/EStructuralFeature/transient : true,
  ecore/ETypedElement/eType : oid#ecore/EClass(
    '#//Table),
  ecore/ETypedElement/lowerBound : 1,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/required : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : 1
>
<
  oid#ecore/EReference(
    '#//ForeignKey/column)
  :
  ecore/EReference
  |
  ecore/EModelElement/eAnnotations,
  ecore/EReference/container,
  ecore/EReference/containment,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/transient,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/lowerBound,
  ecore/ETypedElement/required,
  ecore/ENamedElement/name : "column",
  ecore/EReference/eOpposite : oid#ecore/EReference(
    '#//Column/foreignKey),
  ecore/EReference/eReferenceType : oid#ecore/EClass(
    '#//Column),
  ecore/EReference/resolveProxies : true,
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//ForeignKey),
  ecore/ETypedElement/eType : oid#ecore/EClass(
    '#//Column),
  ecore/ETypedElement/many : true,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : -1
>
<
  oid#ecore/EReference(
    '#//ForeignKey/owner)
  :
  ecore/EReference
  |
  ecore/EModelElement/eAnnotations,
  ecore/EReference/containment,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/many,
  ecore/ENamedElement/name : "owner",
  ecore/EReference/container : true,
  ecore/EReference/eOpposite : oid#ecore/EReference(
    '#//Table/foreignKey),
  ecore/EReference/eReferenceType : oid#ecore/EClass(
    '#//Table),
  ecore/EReference/resolveProxies : true,
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//ForeignKey),
  ecore/EStructuralFeature/transient : true,
  ecore/ETypedElement/eType : oid#ecore/EClass(
    '#//Table),
  ecore/ETypedElement/lowerBound : 1,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/required : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : 1
>
<
  oid#ecore/EReference(
    '#//ForeignKey/refersTo)
  :
  ecore/EReference
  |
  ecore/EModelElement/eAnnotations,
  ecore/EReference/container,
  ecore/EReference/containment,
  ecore/EReference/eOpposite,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/transient,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/many,
  ecore/ENamedElement/name : "refersTo",
  ecore/EReference/eReferenceType : oid#ecore/EClass(
    '#//Key),
  ecore/EReference/resolveProxies : true,
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//ForeignKey),
  ecore/ETypedElement/eType : oid#ecore/EClass(
    '#//Key),
  ecore/ETypedElement/lowerBound : 1,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/required : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : 1
>
<
  oid#ecore/EReference(
    '#//Key/column)
  :
  ecore/EReference
  |
  ecore/EModelElement/eAnnotations,
  ecore/EReference/container,
  ecore/EReference/containment,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/transient,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/lowerBound,
  ecore/ETypedElement/required,
  ecore/ENamedElement/name : "column",
  ecore/EReference/eOpposite : oid#ecore/EReference(
    '#//Column/key),
  ecore/EReference/eReferenceType : oid#ecore/EClass(
    '#//Column),
  ecore/EReference/resolveProxies : true,
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//Key),
  ecore/ETypedElement/eType : oid#ecore/EClass(
    '#//Column),
  ecore/ETypedElement/many : true,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : -1
>
<
  oid#ecore/EReference(
    '#//Key/owner)
  :
  ecore/EReference
  |
  ecore/EModelElement/eAnnotations,
  ecore/EReference/containment,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/many,
  ecore/ENamedElement/name : "owner",
  ecore/EReference/container : true,
  ecore/EReference/eOpposite : oid#ecore/EReference(
    '#//Table/key),
  ecore/EReference/eReferenceType : oid#ecore/EClass(
    '#//Table),
  ecore/EReference/resolveProxies : true,
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//Key),
  ecore/EStructuralFeature/transient : true,
  ecore/ETypedElement/eType : oid#ecore/EClass(
    '#//Table),
  ecore/ETypedElement/lowerBound : 1,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/required : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : 1
>
<
  oid#ecore/EReference(
    '#//Schema/tables)
  :
  ecore/EReference
  |
  ecore/EModelElement/eAnnotations,
  ecore/EReference/container,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/transient,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/lowerBound,
  ecore/ETypedElement/required,
  ecore/ENamedElement/name : "tables",
  ecore/EReference/containment : true,
  ecore/EReference/eOpposite : oid#ecore/EReference(
    '#//Table/schema),
  ecore/EReference/eReferenceType : oid#ecore/EClass(
    '#//Table),
  ecore/EReference/resolveProxies : true,
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//Schema),
  ecore/ETypedElement/eType : oid#ecore/EClass(
    '#//Table),
  ecore/ETypedElement/many : true,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : -1
>
<
  oid#ecore/EReference(
    '#//Table/column)
  :
  ecore/EReference
  |
  ecore/EModelElement/eAnnotations,
  ecore/EReference/container,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/transient,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/lowerBound,
  ecore/ETypedElement/required,
  ecore/ENamedElement/name : "column",
  ecore/EReference/containment : true,
  ecore/EReference/eOpposite : oid#ecore/EReference(
    '#//Column/owner),
  ecore/EReference/eReferenceType : oid#ecore/EClass(
    '#//Column),
  ecore/EReference/resolveProxies : true,
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//Table),
  ecore/ETypedElement/eType : oid#ecore/EClass(
    '#//Column),
  ecore/ETypedElement/many : true,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : -1
>
<
  oid#ecore/EReference(
    '#//Table/foreignKey)
  :
  ecore/EReference
  |
  ecore/EModelElement/eAnnotations,
  ecore/EReference/container,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/transient,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/lowerBound,
  ecore/ETypedElement/required,
  ecore/ENamedElement/name : "foreignKey",
  ecore/EReference/containment : true,
  ecore/EReference/eOpposite : oid#ecore/EReference(
    '#//ForeignKey/owner),
  ecore/EReference/eReferenceType : oid#ecore/EClass(
    '#//ForeignKey),
  ecore/EReference/resolveProxies : true,
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//Table),
  ecore/ETypedElement/eType : oid#ecore/EClass(
    '#//ForeignKey),
  ecore/ETypedElement/many : true,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : -1
>
<
  oid#ecore/EReference(
    '#//Table/key)
  :
  ecore/EReference
  |
  ecore/EModelElement/eAnnotations,
  ecore/EReference/container,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/transient,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/lowerBound,
  ecore/ETypedElement/required,
  ecore/ENamedElement/name : "key",
  ecore/EReference/containment : true,
  ecore/EReference/eOpposite : oid#ecore/EReference(
    '#//Key/owner),
  ecore/EReference/eReferenceType : oid#ecore/EClass(
    '#//Key),
  ecore/EReference/resolveProxies : true,
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//Table),
  ecore/ETypedElement/eType : oid#ecore/EClass(
    '#//Key),
  ecore/ETypedElement/many : true,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : -1
>
<
  oid#ecore/EReference(
    '#//Table/schema)
  :
  ecore/EReference
  |
  ecore/EModelElement/eAnnotations,
  ecore/EReference/containment,
  ecore/EStructuralFeature/defaultValueLiteral,
  ecore/EStructuralFeature/derived,
  ecore/EStructuralFeature/unsettable,
  ecore/EStructuralFeature/volatile,
  ecore/ETypedElement/many,
  ecore/ENamedElement/name : "schema",
  ecore/EReference/container : true,
  ecore/EReference/eOpposite : oid#ecore/EReference(
    '#//Schema/tables),
  ecore/EReference/eReferenceType : oid#ecore/EClass(
    '#//Schema),
  ecore/EReference/resolveProxies : true,
  ecore/EStructuralFeature/changeable : true,
  ecore/EStructuralFeature/defaultValue : "0",
  ecore/EStructuralFeature/eContainingClass : oid#ecore/EClass(
    '#//Table),
  ecore/EStructuralFeature/transient : true,
  ecore/ETypedElement/eType : oid#ecore/EClass(
    '#//Schema),
  ecore/ETypedElement/lowerBound : 1,
  ecore/ETypedElement/ordered : true,
  ecore/ETypedElement/required : true,
  ecore/ETypedElement/unique : true,
  ecore/ETypedElement/upperBound : 1
> >>
```

Appendix C

The RDBMS theory

```
mod mod#rdbms/RDataType is
  sorts rdbms/RDataType .
  op null#rdbms/RDataType : -> [rdbms/RDataType] .
  op rdbms/RDataType/BOOLEAN : -> rdbms/RDataType .
  op rdbms/RDataType/DATE : -> rdbms/RDataType .
  op rdbms/RDataType/DECIMAL : -> rdbms/RDataType .
  op rdbms/RDataType/NUMBER : -> rdbms/RDataType .
  op rdbms/RDataType/VARCHAR : -> rdbms/RDataType .
endm

view rdbms/RDataType from TRIV to mod#rdbms/RDataType is
  sort Elt to rdbms/RDataType .
endv

mod mod#rdbms is
  including OCL-DATATYPE-COLLECTIONS .
  protecting BOOL .
  including OCL-COLLECTIONS{rdbms/RDataType} * (op empty-bag to
    empty-bag#rdbms/RDataType, op empty-orderedset to
    empty-orderedset#rdbms/RDataType, op empty-sequence to
    empty-sequence#rdbms/RDataType, op empty-set to empty-set#rdbms/RDataType)
    .
  sorts Cid#rdbms EObject#rdbms Oid#rdbms Property#rdbms PropertySet#rdbms
    oid#rdbms/Column oid#rdbms/ForeignKey oid#rdbms/Key oid#rdbms/RModelEment
    oid#rdbms/Schema oid#rdbms/Table rdbms/Column rdbms/ForeignKey rdbms/Key
    rdbms/RModelEment rdbms/Schema rdbms/Table .
  subsorts Oid#rdbms < AbstractOid .
  subsorts Property#rdbms < PropertySet#rdbms .
  subsorts oid#rdbms/Column < oid#rdbms/RModelEment .
  subsorts oid#rdbms/ForeignKey < oid#rdbms/RModelEment .
  subsorts oid#rdbms/Key < oid#rdbms/RModelEment .
  subsorts oid#rdbms/RModelEment < Oid#rdbms .
  subsorts oid#rdbms/Schema < oid#rdbms/RModelEment .
  subsorts oid#rdbms/Table < oid#rdbms/RModelEment .
  subsorts rdbms/Column < rdbms/RModelEment .
  subsorts rdbms/ForeignKey < rdbms/RModelEment .
  subsorts rdbms/Key < rdbms/RModelEment .
  subsorts rdbms/RModelEment < Cid#rdbms .
  subsorts rdbms/Schema < rdbms/RModelEment .
  subsorts rdbms/Table < rdbms/RModelEment .
  op class : EObject#rdbms -> Cid#rdbms .
  op getPropertySet : EObject#rdbms -> PropertySet#rdbms .
  op noneProperty#rdbms : -> PropertySet#rdbms .
  op nullEObject#rdbms : -> [EObject#rdbms] .
  op nullOid#rdbms : -> [Oid#rdbms] .
  op oid : EObject#rdbms -> Oid#rdbms .
  op <_:_|_> : Oid#rdbms Cid#rdbms PropertySet#rdbms -> EObject#rdbms [obj ctor
    format (d n++i ni ni ni ni n--i d)] .
  op _‘,_ : PropertySet#rdbms PropertySet#rdbms -> PropertySet#rdbms [assoc
    comm id: noneProperty#rdbms ctor format (d d ni d)] .
  op oid#rdbms/Column : Qid -> oid#rdbms/Column [ctor] .
  op oid#rdbms/ForeignKey : Qid -> oid#rdbms/ForeignKey [ctor] .
  op oid#rdbms/Key : Qid -> oid#rdbms/Key [ctor] .
  op oid#rdbms/RModelEment : Qid -> oid#rdbms/RModelEment [ctor] .
  op oid#rdbms/Schema : Qid -> oid#rdbms/Schema [ctor] .
  op oid#rdbms/Table : Qid -> oid#rdbms/Table [ctor] .
  op rdbms/Column : -> rdbms/Column [ctor] .
  op rdbms/Column/foreignKey/0 : -> Property#rdbms [ctor] .
  op rdbms/Column/foreignKey‘:_ : OrderedSet{Oid} -> Property#rdbms [ctor] .
  op rdbms/Column/key/0 : -> Property#rdbms [ctor] .
  op rdbms/Column/key`:_ : OrderedSet{Oid} -> Property#rdbms [ctor] .
  op rdbms/Column/nnv/0 : -> Property#rdbms [ctor] .
  op rdbms/Column/nnv`:_ : Bool -> Property#rdbms [ctor] .
  op rdbms/Column/owner/0 : -> Property#rdbms [ctor] .
  op rdbms/Column/owner`:_ : Oid#rdbms -> Property#rdbms [ctor] .
  op rdbms/Column/type/0 : -> Property#rdbms [ctor] .
  op rdbms/Column/type`:_ : rdbms/RDataType -> Property#rdbms [ctor] .
  op rdbms/ForeignKey : -> rdbms/ForeignKey [ctor] .
  op rdbms/ForeignKey/column/0 : -> Property#rdbms [ctor] .
  op rdbms/ForeignKey/column`:_ : OrderedSet{Oid} -> Property#rdbms [ctor] .
  op rdbms/ForeignKey/owner/0 : -> Property#rdbms [ctor] .
  op rdbms/ForeignKey/owner`:_ : Oid#rdbms -> Property#rdbms [ctor] .
  op rdbms/ForeignKey/refersTo/0 : -> Property#rdbms [ctor] .
  op rdbms/ForeignKey/refersTo`:_ : Oid#rdbms -> Property#rdbms [ctor] .
  op rdbms/Key : -> rdbms/Key [ctor] .
  op rdbms/Key/column/0 : -> Property#rdbms [ctor] .
  op rdbms/Key/column`:_ : OrderedSet{Oid} -> Property#rdbms [ctor] .
  op rdbms/Key/owner/0 : -> Property#rdbms [ctor] .
  op rdbms/Key/owner`:_ : Oid#rdbms -> Property#rdbms [ctor] .
  op rdbms/RModelEment : -> rdbms/RModelEment [ctor] .
  op rdbms/RModelEment/name/0 : -> Property#rdbms [ctor] .
  op rdbms/RModelEment/name`:_ : String -> Property#rdbms [ctor] .
  op rdbms/Schema : -> rdbms/Schema [ctor] .
  op rdbms/Schema/tables/0 : -> Property#rdbms [ctor] .
  op rdbms/Schema/tables`:_ : OrderedSet{Oid} -> Property#rdbms [ctor] .
  op rdbms/Table : -> rdbms/Table [ctor] .
  op rdbms/Table/column/0 : -> Property#rdbms [ctor] .
  op rdbms/Table/column`:_ : OrderedSet{Oid} -> Property#rdbms [ctor] .
  op rdbms/Table/foreignKey/0 : -> Property#rdbms [ctor] .
  op rdbms/Table/foreignKey`:_ : OrderedSet{Oid} -> Property#rdbms [ctor] .
  op rdbms/Table/key/0 : -> Property#rdbms [ctor] .
  op rdbms/Table/key`:_ : OrderedSet{Oid} -> Property#rdbms [ctor] .
  op rdbms/Table/schema/0 : -> Property#rdbms [ctor] .
  op rdbms/Table/schema`:_ : Oid#rdbms -> Property#rdbms [ctor] .
  eq class (< Oid:Oid#rdbms : CID:Cid#rdbms | PS:PropertySet#rdbms >) =
    CID:Cid#rdbms .
  eq getPropertySet (< Oid:Oid#rdbms : CID:Cid#rdbms | PS:PropertySet#rdbms >)
    = PS:PropertySet#rdbms .
  eq oid (< Oid:Oid#rdbms : CID:Cid#rdbms | PS:PropertySet#rdbms >) =
    Oid:Oid#rdbms .
endm

view rdbms from TH-EOBJECT to mod#rdbms is
  sort Cid to Cid#rdbms .
  sort EObject to EObject#rdbms .
  sort Oid to Oid#rdbms .
  sort Property to Property#rdbms .
  sort PropertySet to PropertySet#rdbms .
  op noneProperty to noneProperty#rdbms .
  op nullEObject to nullEObject#rdbms .
  op nullOid to nullOid#rdbms .
endv
```

Appendix D

The *rsPerson* relational schema definition

```
<< <
  oid#rdbms/Column(''#//@tables.0/@column.0)
  :
  rdbms/Column
  |
  rdbms/Column/foreignKey/0,
  rdbms/Column/key/0,
  rdbms/Column/nnv : true,
  rdbms/Column/owner : oid#rdbms/Table(''#//@tables.0),
  rdbms/Column/type : rdbms/RDataType/VARCHAR,
  rdbms/RModelEment/name : "name"
>
<
  oid#rdbms/Column(''#//@tables.0/@column.1)
  :
  rdbms/Column
  |
  rdbms/Column/foreignKey/0,
  rdbms/Column/key/0,
  rdbms/Column/nnv/0,
  rdbms/Column/owner : oid#rdbms/Table(''#//@tables.0),
  rdbms/Column/type : rdbms/RDataType/NUMBER,
  rdbms/RModelEment/name : "age"
>
<
  oid#rdbms/Column(''#//@tables.0/@column.2)
  :
  rdbms/Column
  |
  rdbms/Column/foreignKey/0,
  rdbms/Column/key : OrderedSet{oid#rdbms/Key(''#//@tables.0/@key.0)},
  rdbms/Column/nnv : true,
  rdbms/Column/owner : oid#rdbms/Table(''#//@tables.0),
  rdbms/Column/type : rdbms/RDataType/VARCHAR,
  rdbms/RModelEment/name : "person_PK"
>
<
  oid#rdbms/Column(''#//@tables.1/@column.0)
  :
  rdbms/Column
  |
  rdbms/Column/foreignKey/0,
  rdbms/Column/key/0,
  rdbms/Column/nnv : true,
  rdbms/Column/owner : oid#rdbms/Table(''#//@tables.1),
  rdbms/Column/type : rdbms/RDataType/DATE,
  rdbms/RModelEment/name : "date"
>
<
  oid#rdbms/Column(''#//@tables.1/@column.1)
  :
  rdbms/Column
  |
  rdbms/Column/foreignKey/0,
  rdbms/Column/key/0,
  rdbms/Column/nnv/0,
  rdbms/Column/owner : oid#rdbms/Table(''#//@tables.1),
  rdbms/Column/type : rdbms/RDataType/DECIMAL,
  rdbms/RModelEment/name : "cost"
>
<
  oid#rdbms/Column(''#//@tables.1/@column.2)
  :
  rdbms/Column
  |
  rdbms/Column/foreignKey/0,
  rdbms/Column/key : OrderedSet{oid#rdbms/Key(''#//@tables.1/@key.0)},
  rdbms/Column/nnv : true,
  rdbms/Column/owner : oid#rdbms/Table(''#//@tables.1),
  rdbms/Column/type : rdbms/RDataType/VARCHAR,
  rdbms/RModelEment/name : "invoice_PK"
>
<
  oid#rdbms/Column(''#//@tables.1/@column.3)
  :
  rdbms/Column
  |
  rdbms/Column/key/0,
  rdbms/Column/foreignKey : OrderedSet{oid#rdbms/ForeignKey(
    ''#//@tables.1/@foreignKey.0)},
  rdbms/Column/nnv : true,
  rdbms/Column/owner : oid#rdbms/Table(''#//@tables.1),
  rdbms/Column/type : rdbms/RDataType/VARCHAR,
  rdbms/RModelEment/name : "person_FK"
>
<
  oid#rdbms/Column(''#//@tables.2/@column.0)
  :
  rdbms/Column
  |
  rdbms/Column/foreignKey/0,
  rdbms/Column/key/0,
  rdbms/Column/nnv : true,
  rdbms/Column/owner : oid#rdbms/Table(''#//@tables.2),
  rdbms/Column/type : rdbms/RDataType/VARCHAR,
  rdbms/RModelEment/name : "name"
>
<
  oid#rdbms/Column(''#//@tables.2/@column.1)
  :
  rdbms/Column
  |
  rdbms/Column/foreignKey/0,
  rdbms/Column/key/0,
  rdbms/Column/nnv : true,
  rdbms/Column/owner : oid#rdbms/Table(''#//@tables.2),
  rdbms/Column/type : rdbms/RDataType/DECIMAL,
  rdbms/RModelEment/name : "price"
>
<
  oid#rdbms/Column(''#//@tables.2/@column.2)
  :
  rdbms/Column
  |
  rdbms/Column/foreignKey/0,
  rdbms/Column/nnv/0,
  rdbms/Column/key : OrderedSet{oid#rdbms/Key(''#//@tables.2/@key.0)},
  rdbms/Column/owner : oid#rdbms/Table(''#//@tables.2),
  rdbms/Column/type : rdbms/RDataType/VARCHAR,
  rdbms/RModelEment/name : "item_PK"
>
<
  oid#rdbms/Column(''#//@tables.2/@column.3)
  :
  rdbms/Column
  |
  rdbms/Column/key/0,
  rdbms/Column/foreignKey : OrderedSet{oid#rdbms/ForeignKey(
    ''#//@tables.2/@foreignKey.0)},
  rdbms/Column/nnv : true,
  rdbms/Column/owner : oid#rdbms/Table(''#//@tables.2),
  rdbms/Column/type : rdbms/RDataType/VARCHAR,
  rdbms/RModelEment/name : "invoice_FK"
>
<
  oid#rdbms/ForeignKey(''#//@tables.1/@foreignKey.0)
  :
  rdbms/ForeignKey
  |
  rdbms/ForeignKey/column : OrderedSet{oid#rdbms/Column(
    ''#//@tables.1/@column.3)},
  rdbms/ForeignKey/owner : oid#rdbms/Table(''#//@tables.1),
  rdbms/ForeignKey/refersTo : oid#rdbms/Key(''#//@tables.0/@key.0),
  rdbms/RModelEment/name : "Invoice_Person_FK"
>
<
  oid#rdbms/ForeignKey(''#//@tables.2/@foreignKey.0)
  :
  rdbms/ForeignKey
  |
  rdbms/ForeignKey/column : OrderedSet{oid#rdbms/Column(
    ''#//@tables.2/@column.3)},
  rdbms/ForeignKey/owner : oid#rdbms/Table(''#//@tables.2),
  rdbms/ForeignKey/refersTo : oid#rdbms/Key(''#//@tables.1/@key.0),
  rdbms/RModelEment/name : "Item_Invoice_FK"
>
<
  oid#rdbms/Key(''#//@tables.0/@key.0)
  :
  rdbms/Key
  |
  rdbms/Key/column : OrderedSet{oid#rdbms/Column(''#//@tables.0/@column.2)},
  rdbms/Key/owner : oid#rdbms/Table(''#//@tables.0),
  rdbms/RModelEment/name : "Person_PK"
>
<
  oid#rdbms/Key(''#//@tables.1/@key.0)
  :
  rdbms/Key
  |
  rdbms/Key/column : OrderedSet{oid#rdbms/Column(''#//@tables.1/@column.2)},
  rdbms/Key/owner : oid#rdbms/Table(''#//@tables.1),
  rdbms/RModelEment/name : "Invoice_PK"
>
<
  oid#rdbms/Key(''#//@tables.2/@key.0)
  :
  rdbms/Key
  |
  rdbms/Key/column : OrderedSet{oid#rdbms/Column(''#//@tables.2/@column.2)},
  rdbms/Key/owner : oid#rdbms/Table(''#//@tables.2),
  rdbms/RModelEment/name : "Item_PK"
>
<
  oid#rdbms/Schema('platform:/resource/metamodels/qvtrdbms/rsInvoice.xmi#/)
  :
  rdbms/Schema
  |
  rdbms/RModelEment/name : "rsInvoice",
  rdbms/Schema/tables : OrderedSet{oid#rdbms/Table(''#//@tables.0) ::
    oid#rdbms/Table(''#//@tables.1) :: oid#rdbms/Table(''#//@tables.2)}
>
<
  oid#rdbms/Table(''#//@tables.0)
  :
  rdbms/Table
  |
  rdbms/Table/foreignKey/0,
  rdbms/RModelEment/name : "Person",
  rdbms/Table/column : OrderedSet{oid#rdbms/Column(''#//@tables.0/@column.0) ::
    oid#rdbms/Column(''#//@tables.0/@column.1) :: oid#rdbms/Column(
    ''#//@tables.0/@column.2)},
  rdbms/Table/key : OrderedSet{oid#rdbms/Key(''#//@tables.0/@key.0)},
  rdbms/Table/schema : oid#rdbms/Schema(
    'platform:/resource/metamodels/qvtrdbms/rsInvoice.xmi#/)
>
<
  oid#rdbms/Table(''#//@tables.1)
  :
  rdbms/Table
  |
  rdbms/RModelEment/name : "Invoice",
  rdbms/Table/column : OrderedSet{oid#rdbms/Column(''#//@tables.1/@column.0) ::
    oid#rdbms/Column(''#//@tables.1/@column.1) :: oid#rdbms/Column(
    ''#//@tables.1/@column.2) :: oid#rdbms/Column(''#//@tables.1/@column.3)},
  rdbms/Table/foreignKey : OrderedSet{oid#rdbms/ForeignKey(
    ''#//@tables.1/@foreignKey.0)},
  rdbms/Table/key : OrderedSet{oid#rdbms/Key(''#//@tables.1/@key.0)},
  rdbms/Table/schema : oid#rdbms/Schema(
    'platform:/resource/metamodels/qvtrdbms/rsInvoice.xmi#/)
>
<
  oid#rdbms/Table(''#//@tables.2)
  :
  rdbms/Table
  |
  rdbms/RModelEment/name : "Item",
  rdbms/Table/column : OrderedSet{oid#rdbms/Column(''#//@tables.2/@column.0) ::
    oid#rdbms/Column(''#//@tables.2/@column.1) :: oid#rdbms/Column(
    ''#//@tables.2/@column.2) :: oid#rdbms/Column(''#//@tables.2/@column.3)},
  rdbms/Table/foreignKey : OrderedSet{oid#rdbms/ForeignKey(
    ''#//@tables.2/@foreignKey.0)},
  rdbms/Table/key : OrderedSet{oid#rdbms/Key(''#//@tables.2/@key.0)},
  rdbms/Table/schema : oid#rdbms/Schema(
    'platform:/resource/metamodels/qvtrdbms/rsInvoice.xmi#/)
> >>
```

Appendix E

The metarepresented *rsPerson* relational schema definition

```
<< < metaOid("oid#rdbms/Column", ''#//@tables.0/@column.0) :
  ecore/EObject |
  property :
    "foreignKey"
  ,
  property :
    "key"
  ,
  class : "rdbms/Column",
  (property :
    "name"
    =
    "name"
  ),
  (property :
    "nnv"
    =
    true
  ),
  (property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.0)
  ),
  property :
    "type"
    =
    metaEEnumLiteral("rdbms/RDataType/VARCHAR")

>
< metaOid("oid#rdbms/Column", ''#//@tables.0/@column.1) :
  ecore/EObject |
  property :
    "foreignKey"
  ,
  property :
    "key"
  ,
  property :
    "nnv"
  ,
  class : "rdbms/Column",
  (property :
    "name"
    =
    "age"
  ),
  (property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.0)
  ),
  property :
    "type"
    =
    metaEEnumLiteral("rdbms/RDataType/NUMBER")

>
< metaOid("oid#rdbms/Column", ''#//@tables.0/@column.2) :
  ecore/EObject |
  property :
    "foreignKey"
  ,
  class : "rdbms/Column",
  (property :
    "name"
    =
    "person_PK"
  ),
  (property :
    "nnv"
    =
    true
  ),
  (property :
    "key"
    =
    OrderedSet{metaOid("oid#rdbms/Key",
    ''#//@tables.0/@key.0)}
  ),
  (property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.0)
  ),
  property :
    "type"
    =
    metaEEnumLiteral("rdbms/RDataType/VARCHAR")

>
< metaOid("oid#rdbms/Column", ''#//@tables.1/@column.0) :
  ecore/EObject |
  property :
    "foreignKey"
  ,
  property :
    "key"
  ,
  class : "rdbms/Column",
  (property :
    "name"
    =
    "date"
  ),
  (property :
    "nnv"
    =
    true
  ),
  (property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.1)
  ),
  property :
    "type"
    =
    metaEEnumLiteral("rdbms/RDataType/DATE")

>
< metaOid("oid#rdbms/Column", ''#//@tables.1/@column.1) :
  ecore/EObject |
  property :
    "foreignKey"
  ,
  property :
    "key"
  ,
  property :
    "nnv"
  ,
  class : "rdbms/Column",
  (property :
    "name"
    =
    "cost"
  ),
  (property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.1)
  ),
  property :
    "type"
    =
    metaEEnumLiteral("rdbms/RDataType/DECIMAL")

>
< metaOid("oid#rdbms/Column", ''#//@tables.1/@column.2) :
  ecore/EObject |
  property :
    "foreignKey"
  ,
  class : "rdbms/Column",
  (property :
    "name"
    =
    "invoice_PK"
  ),
  (property :
    "nnv"
    =
    true
  ),
  (property :
    "key"
    =
    OrderedSet{metaOid("oid#rdbms/Key",
    ''#//@tables.1/@key.0)}
  ),
  (property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.1)
  ),
  property :
    "type"
    =
    metaEEnumLiteral("rdbms/RDataType/VARCHAR")

>
< metaOid("oid#rdbms/Column", ''#//@tables.1/@column.3) :
  ecore/EObject |
  property :
    "key"
  ,
  class : "rdbms/Column",
  (property :
    "name"
    =
    "person_FK"
  ),
  (property :
    "nnv"
    =
    true
  ),
  (property :
    "foreignKey"
    =
    OrderedSet{metaOid("oid#rdbms/ForeignKey", ''#//@tables.1/@foreignKey.0)}
  ),
  (property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.1)
  ),
  property :
    "type"
    =
    metaEEnumLiteral("rdbms/RDataType/VARCHAR")

>
< metaOid("oid#rdbms/Column", ''#//@tables.2/@column.0) :
  ecore/EObject |
  property :
    "foreignKey"
  ,
  property :
    "key"
  ,
  class : "rdbms/Column",
  (property :
    "name"
    =
    "name"
  ),
  (property :
    "nnv"
    =
    true
  ),
  (property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.2)
  ),
  property :
    "type"
    =
    metaEEnumLiteral("rdbms/RDataType/VARCHAR")

>
< metaOid("oid#rdbms/Column", ''#//@tables.2/@column.1) :
  ecore/EObject |
  property :
    "foreignKey"
  ,
  property :
    "key"
  ,
  class : "rdbms/Column",
  (property :
    "name"
    =
    "price"
  ),
  (property :
    "nnv"
    =
    true
  ),
  (property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.2)
  ),
  property :
    "type"
    =
    metaEEnumLiteral("rdbms/RDataType/DECIMAL")

>
< metaOid("oid#rdbms/Column", ''#//@tables.2/@column.2) :
  ecore/EObject |
  property :
    "foreignKey"
  ,
  property :
    "nnv"
  ,
  class : "rdbms/Column",
  (property :
    "name"
    =
    "item_PK"
  ),
  (property :
    "key"
    =
    OrderedSet{metaOid("oid#rdbms/Key",
    ''#//@tables.2/@key.0)}
  ),
  (property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.2)
  ),
  property :
    "type"
    =
    metaEEnumLiteral("rdbms/RDataType/VARCHAR")

>
< metaOid("oid#rdbms/Column", ''#//@tables.2/@column.3) :
  ecore/EObject |
  property :
    "key"
  ,
  class : "rdbms/Column",
  (property :
    "name"
    =
    "invoice_FK"
  ),
  (property :
    "nnv"
    =
    true
  ),
  (property :
    "foreignKey"
    =
    OrderedSet{metaOid("oid#rdbms/ForeignKey", ''#//@tables.2/@foreignKey.0)}
  ),
  (property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.2)
  ),
  property :
    "type"
    =
    metaEEnumLiteral("rdbms/RDataType/VARCHAR")

>
< metaOid("oid#rdbms/ForeignKey", ''#//@tables.1/@foreignKey.0) :
  ecore/EObject |
  class : "rdbms/ForeignKey",
  (property :
    "name"
    =
    "Invoice_Person_FK"
  ),
  (property :
    "column"
    =
    OrderedSet{metaOid("oid#rdbms/Column", ''#//@tables.1/@column.3)}
  ),
  (property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.1)
  ),
  property :
    "refersTo"
    =
    metaOid("oid#rdbms/Key",
    ''#//@tables.0/@key.0)

>
< metaOid("oid#rdbms/ForeignKey", ''#//@tables.2/@foreignKey.0) :
  ecore/EObject |
  class : "rdbms/ForeignKey",
  (property :
    "name"
    =
    "Item_Invoice_FK"
  ),
  (property :
    "column"
    =
    OrderedSet{metaOid("oid#rdbms/Column", ''#//@tables.2/@column.3)}
  ),
  (property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.2)
  ),
  property :
    "refersTo"
    =
    metaOid("oid#rdbms/Key",
    ''#//@tables.1/@key.0)

>
< metaOid("oid#rdbms/Key",
    ''#//@tables.0/@key.0) :
  ecore/EObject |
  class : "rdbms/Key",
  (property :
    "name"
    =
    "Person_PK"
  ),
  (property :
    "column"
    =
    OrderedSet{metaOid("oid#rdbms/Column", ''#//@tables.0/@column.2)}
  ),
  property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.0)

>
< metaOid("oid#rdbms/Key",
    ''#//@tables.1/@key.0) :
  ecore/EObject |
  class : "rdbms/Key",
  (property :
    "name"
    =
    "Invoice_PK"
  ),
  (property :
    "column"
    =
    OrderedSet{metaOid("oid#rdbms/Column", ''#//@tables.1/@column.2)}
  ),
  property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.1)

>
< metaOid("oid#rdbms/Key",
    ''#//@tables.2/@key.0) :
  ecore/EObject |
  class : "rdbms/Key",
  (property :
    "name"
    =
    "Item_PK"
  ),
  (property :
    "column"
    =
    OrderedSet{metaOid("oid#rdbms/Column", ''#//@tables.2/@column.2)}
  ),
  property :
    "owner"
    =
    metaOid("oid#rdbms/Table",
    ''#//@tables.2)

>
< metaOid("oid#rdbms/Schema",
    'platform:/resource/metamodels/qvtrdbms/rsInvoice.xmi#/) :
  ecore/EObject |
  class : "rdbms/Schema",
  (property :
    "name"
    =
    "rsInvoice"
  ),
  property :
    "tables"
    =
    OrderedSet{metaOid("oid#rdbms/Table",
    ''#//@tables.0) ::
    metaOid("oid#rdbms/Table",
    ''#//@tables.1) ::
    metaOid("oid#rdbms/Table",
    ''#//@tables.2)}

>
< metaOid("oid#rdbms/Table",
    ''#//@tables.0) :
  ecore/EObject |
  property :
    "foreignKey"
  ,
  class : "rdbms/Table",
  (property :
    "name"
    =
    "Person"
  ),
  (property :
    "column"
    =
    OrderedSet{metaOid("oid#rdbms/Column", ''#//@tables.0/@column.0) :: metaOid("oid#rdbms/Column", ''#//@tables.0/@column.1) :: metaOid
    "oid#rdbms/Column", ''#//@tables.0/@column.2)}
  ),
  (property :
    "key"
    =
    OrderedSet{metaOid("oid#rdbms/Key",
    ''#//@tables.0/@key.0)}
  ),
  property :
    "schema"
    =
    metaOid("oid#rdbms/Schema",
    'platform:/resource/metamodels/qvtrdbms/rsInvoice.xmi#/)

>
< metaOid("oid#rdbms/Table",
    ''#//@tables.1) :
  ecore/EObject |
  class : "rdbms/Table",
  (property :
    "name"
    =
    "Invoice"
  ),
  (property :
    "column"
    =
    OrderedSet{metaOid("oid#rdbms/Column", ''#//@tables.1/@column.0) :: metaOid("oid#rdbms/Column", ''#//@tables.1/@column.1) :: metaOid
    "oid#rdbms/Column", ''#//@tables.1/@column.2) :: metaOid("oid#rdbms/Column", ''#//@tables.1/@column.3)}
  ),
  (property :
    "foreignKey"
    =
    OrderedSet{metaOid("oid#rdbms/ForeignKey", ''#//@tables.1/@foreignKey.0)}
  ),
  (property :
    "key"
    =
    OrderedSet{metaOid("oid#rdbms/Key",
    ''#//@tables.1/@key.0)}
  ),
  property :
    "schema"
    =
    metaOid("oid#rdbms/Schema",
    'platform:/resource/metamodels/qvtrdbms/rsInvoice.xmi#/)

>
< metaOid("oid#rdbms/Table",
    ''#//@tables.2) :
  ecore/EObject |
  class : "rdbms/Table",
  (property :
    "name"
    =
    "Item"
  ),
  (property :
    "column"
    =
    OrderedSet{metaOid("oid#rdbms/Column", ''#//@tables.2/@column.0) ::
    metaOid("oid#rdbms/Column", ''#//@tables.2/@column.1) :: metaOid(
    "oid#rdbms/Column", ''#//@tables.2/@column.2) ::
    metaOid("oid#rdbms/Column", ''#//@tables.2/@column.3)}
  ),
  (property :
    "foreignKey"
    =
    OrderedSet{metaOid("oid#rdbms/ForeignKey", ''#//@tables.2/@foreignKey.0)}
  ),
  (property :
    "key"
    =
    OrderedSet{metaOid("oid#rdbms/Key",
    ''#//@tables.2/@key.0)}
  ),
  property :
    "schema"
    =
    metaOid("oid#rdbms/Schema",
    'platform:/resource/metamodels/qvtrdbms/rsInvoice.xmi#/)
> >>
```

Appendix F

Algebraic Specification of OCL Collection Operators

```
fmod OCL-COLLECTIONS{T :: TRIV} is
    pr OCL-COLLECTION-TYPES{T}  .

    *** user functions that return a boolean value
    *** Body: function that manipulates an element of a Collection{T}
    *** BoolBody: function that queries an element of a Collection{T} and returns a boolean value
    sorts Body{T} BoolBody{T} .
    sort IterateBody{T} .

    sort PreConfiguration{T} .
    op nonePreConf : -> PreConfiguration{T} .


    *** VARIABLE SUPPORT FOR THE ENVIRONMENT
    op _=_ : OclVariableName Collection+{T} -> OclVariable .


    *** ****************************************************************
    *** ****************************************************************
    ***
    *** COMMON OPERATORS
    ***
    *** ****************************************************************
    *** ****************************************************************
    op _ocl=_ : Collection+{T} Collection+{T} -> Bool .
    eq (N1 ocl= N2) = N1 == N2 .
    eq (undefN1 ocl= undefN2) =
        not(undefN1 :: Collection+{T}) and not(undefN2 :: Collection+{T}) [owise] .

    op _ocl<>_ : Collection+{T} Collection+{T} -> Bool .
    eq undefN1 ocl<> undefN2 = not(undefN1 ocl= undefN2) .

    op _.`oclIsUndefined : [Collection+{T}] -> Bool .
    eq N:[Collection+{T}] . oclIsUndefined = not(N:[Collection+{T}] :: Collection+{T}) .



    *** ****************************************************************
    *** ****************************************************************
    ***
    *** Collection{T} OPERATIONS
    ***
    *** ****************************************************************
    *** ****************************************************************




    *** ********************************************************************
    *** Collection{T} conversions
    ***
    var E E' : Collection+{T} .
    vars undefN1 undefN2 : Collection+{T} .
    vars N N1 N2 : Collection+{T} .
    vars M M1 M2 M3 M4 M11 M22 : Magma{T} .
    vars OM OM1 OM2 : OrderedMagma{T} .
    vars Col : Collection{T} .
    vars PreConf : PreConfiguration{T} .
    vars Set Set1 Set2 Set11 Set22 : Set{T} .
    vars OSet OSet1 OSet2 OSet11 OSet22 : OrderedSet{T} .
    vars Bag Bag1 Bag2 Bag11 Bag22 : Bag{T} .
    vars Seq Seq1 Seq2 Seq11 Seq22 : Sequence{T} .

    vars i j : Int .

    var B : Body{T} .
    var BB : BoolBody{T} .
    var IF : IterateBody{T} .
    var Env : Environment .
    var acc : OclVariable .
    var MinorOperator : BoolBody{T} .
    var I : Int .
    var SortingCriteria : BoolBody{T} .

    var VName : OclVariableName .




    *** **********************************************************************
    *** **********************************************************************
    *** **********************************************************************
    *** Collection{T} conversions
    ***

    *** asSet(): Set(T)
    op _->`asSet : Collection{T} -> Set{T} [prec 35] .

    *** asOrderedSet(): OrderedSet(T)
    *** only defined for OrderedSet and Sequence
    *** and for collections of basic data types (string, int, float)
    op _->`asOrderedSet : Collection{T} -> OrderedSet{T} [prec 35] .

    *** asBag() : Bag(T)
    op _->`asBag : Collection{T} -> Bag{T} [prec 35] .

    *** asSequence(): Sequence(T)
    *** only defined for OrderedSet and Sequence
    *** and for collections of basic data types (string, int, float)
    op _->`asSequence : Collection{T} -> Sequence{T} [prec 35] .




    *** **********************************************************************
    *** **********************************************************************
    *** **********************************************************************
    *** Collection{T} Operations

    *** size() : Integer
    op _->`size : Collection{T} -> Int . *** size

    *** count(object: T): Integer
    op _->`count`(_`) : Collection{T} Collection{T} -> Int [memo] .

    *** includes(object: T): Boolean
    op _->`includes`(_`) : Collection{T} Collection{T} -> Bool [memo] .

    *** excludes(object: T): Boolean
    op _->`excludes`(_`) : Collection{T} Collection{T} -> Bool [memo] .

    *** includesAll(c2: Collection{T}(T)): Boolean
    op _->`includesAll`(_`) : Collection{T} Collection{T} -> Bool [memo] .

    *** excludesAll(c2: Collection{T}(T)): Boolean
    op _->`excludesAll`(_`) : Collection{T} Collection{T} -> Bool [memo] .


    *** isEmpty(): Boolean
    op _->`isEmpty : Collection{T} -> Bool . ***  isEmpty notEmpty

    *** notEmpty(): Boolean
    op _->`notEmpty : Collection{T} -> Bool . ***  isEmpty notEmpty
```

```
*** sum(): T

*** product(c2: Collection{T}(T2)) : Set(Tuple(first: T, second: T2))

*** union
op _->`union`(_`) : Collection{T} Collection{T} -> Collection{T} [memo] .

*** intersection
op _->`intersection`(_`) : Collection{T} Collection{T} -> Collection{T} [memo] .

*** append(object: T): OrderedSet(T)
op _->`append`(_`) : OrderedSet{T} Collection{T} -> OrderedSet{T} [memo] .
op _->`append`(_`) : OrderedSet{T} Collection{T} -> OrderedSet{T} [memo] .

*** prepend(object: T): OrderedSet(T)
op _->`prepend`(_`) : OrderedSet{T} Collection{T} -> OrderedSet{T} [memo] .
op _->`prepend`(_`) : OrderedSet{T} Collection{T} -> OrderedSet{T} [memo] .

*** ********************************************************************************
*** NOT IN OCL
*** appendCol(c2: OrderedSet(T1)) : OrderedSet(T1) --> not in OCL
op _->`appendCol`(_`) : OrderedSet{T} Collection{T} -> OrderedSet{T} [memo] .
op _->`appendCol`(_`) : OrderedSet{T} Collection{T} -> OrderedSet{T} [memo] .

*** prependCol(c2: OrderedSet(T1)) : OrderedSet(T1)  --> not in OCL
op _->`prependCol`(_`) : OrderedSet{T} Collection{T} -> OrderedSet{T} [memo] .
op _->`prependCol`(_`) : OrderedSet{T} Collection{T} -> OrderedSet{T} [memo] .
*** ********************************************************************************


***  including excluding
op _->`including`(_`) : Collection{T} Collection+{T} -> Collection{T} [memo] .
op _->`excluding`(_`) : Collection{T} Collection+{T} -> Collection{T} [memo] .

*** ********************************************************************************
*** NOT IN OCL
op _->`includingCol`(_`) : Collection{T} Collection{T} -> Collection{T} [memo] .
*** ********************************************************************************


*** first() : T
op _->`first : OrderedSet{T} -> Collection+{T}  .
op _->`first : Sequence{T} -> Collection+{T}  .

*** last() : T
op _->`last : OrderedSet{T} -> Collection+{T}  .
op _->`last : Sequence{T} -> Collection+{T}  .

*** flatten
*** this operator can only flatten a set or a bag into
*** an ordered collection automatically if the collection
*** only contains basic data types (int, float or string)
op _->`flatten : Collection{T} -> Collection{T}  .

*** at(i: Integer) : T
op _->`at`(_`) : OrderedSet{T} Int -> Collection+{T} [memo] .
op _->`at`(_`) : Sequence{T} Int -> Collection+{T} [memo] .

*** indexOf(obj : T) : Integer
op _->`indexOf`(_`) : OrderedSet{T} Collection+{T} -> Int [memo] .
op _->`indexOf`(_`) : Sequence{T} Collection+{T} -> Int [memo] .

*** insertAt(index: Integer, object: T) : OrderedSet(T)
op _->`insertAt`(_;_`) : OrderedSet{T} Int Collection+{T} -> OrderedSet{T}  .
op _->`insertAt`(_;_`) : Sequence{T} Int Collection+{T} -> Sequence{T}  .




*** ****************************************************************
*** ****************************************************************
*** ITERATOR OPERATIONS

op _->`forAll`(_;_;_`) : Collection{T} BoolBody{T} Environment
    PreConfiguration{T} -> Bool  .
op _->`forAll2`(_;_;_`) : Collection{T} BoolBody{T} Environment
    PreConfiguration{T} -> Bool  .
op _->`exists`(_;_;_`) : Collection{T} BoolBody{T} Environment
    PreConfiguration{T} -> Bool  .
op _->`one`(_;_;_`) : Collection{T} BoolBody{T} Environment
    PreConfiguration{T} -> Bool  .
```

```
op _->`isUnique`(_;_;_`) : Collection{T} Body{T} Environment
    PreConfiguration{T} -> Bool  .

op _->`select`(_;_;_`) : Collection{T} BoolBody{T} Environment
    PreConfiguration{T} -> Collection{T}  .
op _->`reject`(_;_;_`) : Collection{T} BoolBody{T} Environment
    PreConfiguration{T} -> Collection{T}  .

*** any can only be used when the boolean body expression returns a collection
*** of int, float or string (otherwise it is not confluent)
op _->`any`(_;_;_`) : Collection{T} BoolBody{T} Environment
    PreConfiguration{T} -> Collection{T}  .

op _->`sortedBy`(_;_;_`) : Collection{T} BoolBody{T} Environment
    PreConfiguration{T} -> Collection{T}  .
op _->`collect`(_;_;_`) : Collection{T} Body{T} Environment
    PreConfiguration{T} -> Collection{T}  .
op _->`collectNested`(_;_;_`) : Collection{T} Body{T} Environment
    PreConfiguration{T} -> Collection{T}  .



*** USER Body{T} FUNCTIONS
*** without parameters
op _._`(_`) : Magma{T} BoolBody{T} PreConfiguration{T} -> Bool  .
op _._`(_`) : OrderedMagma{T} BoolBody{T} PreConfiguration{T} -> Bool  .
op _._`(_`) : Magma{T} Body{T} PreConfiguration{T} -> Collection{T}  .
op _._`(_`) : OrderedMagma{T} Body{T} PreConfiguration{T} -> Collection{T}  .
*** with parameters
op _._`(_;_`) : Magma{T} BoolBody{T} Environment PreConfiguration{T} -> Bool  .
op _._`(_;_`) : OrderedMagma{T} BoolBody{T} Environment PreConfiguration{T} -> Bool  .
op _._`(_;_`) : Magma{T} Body{T} Environment PreConfiguration{T} -> Collection{T}  .
op _._`(_;_`) : OrderedMagma{T} Body{T} Environment PreConfiguration{T}
    -> Collection{T}  .



*** ******************************************************************
*** Collection{T} iterators
***

*** iterate
*** 1: set
*** 2: iterate
*** 3: accumulator. It is a OclVariable: ? T where T can be an String, an Integer, a Set...
*** 4: funcio
*** 5: parameters
*** 6: set -> the whole model
*** This function returns a parameter, i.e. its return value is
*** polymorphic (it can be String, Int, Set...)
op _->`iterate`(_|_;_;_`) : Collection{T} OclVariable IterateBody{T}
    Environment PreConfiguration{T} -> OclVariable [prec 40 gather (E & & & e)] .

*** poly can only be used for constructors and builtins, not for operations
*** 1: Collection+{T}
*** 2: IterateBody
*** 3: accumulator: initial value
*** [4] : OclParameters for the iterate function
*** 5: NodeSet -> the whole model
*** the return type is the same of the accumulator: it is a parameter
*** (integer, string, set, ...)
*** with parameters
op _._`(_;_;_`) : Collection+{T} IterateBody{T} OclVariable
    Environment PreConfiguration{T} -> OclVariable .



*** ******************************************************************
*** ******************************************************************
*** SET
***
*** Duplicates are not allowed: this should be check by the user
*** for the sake of efficiency
***
*** ******************************************************************
*** ******************************************************************



*** *********************************************************************
*** Collection{T} conversions
***
*** asSet(): Set(T)
eq Set -> asSet = Set .

*** asOrderedSet(): OrderedSet(T)
*** including does not take into account uniqueness but if we come from a set,
*** this is not required

*** asBag() : Bag(T)
eq Set{ M } -> asBag = Bag{ M } .
eq empty-set -> asBag = empty-bag .

*** asSequence(): Sequence(T)



*** ********************************************************************
*** Collection{T} OPERATIONS

*** size
eq empty-set -> size = 0 .
eq Set{ N } -> size = 1 .
eq Set{ N , M } -> size = (Set{ M } -> size) + 1 .

*** count
eq empty-set -> count ( N2 ) = 0 .
eq Set{ N1 } -> count ( N1 ) = 1 .
eq Set{ N1 } -> count ( N2 ) = 0 .
eq Set{ N1 , M } -> count ( N1 ) =
    1 + (Set{ M } -> count ( N1 )) .
eq Set{ M } -> count ( N1 ) = 0 [owise] .

*** includes(object: T): Boolean
eq empty-set -> includes ( N ) = false .
eq Set -> includes ( N ) = (Set -> count ( N ) ) > 0 .

*** excludes(object: T): Boolean
eq empty-set -> excludes ( N ) = true .
eq Set -> excludes ( N ) = (Set -> count ( N ) ) == 0 .



*** includesAll -> defined for all Collections

*** excludesAll -> defined for all Collections

*** isEmpty(): Boolean
eq empty-set -> isEmpty = true .
eq Col -> isEmpty = false [owise] .

*** notEmpty(): Boolean
eq empty-set -> notEmpty = false .
eq Col -> notEmpty = true [owise] .

*** sum(): T
*** only for rationals (integers)
*** product(c2: Collection{T}(T2)) : Set(Tuple(first: T, second: T2))


*** ********************************************************************
*** SET OPERATIONS

*** union(s: Set(T)): Set(T) --> only for sorts that have defined ==
*** to apply the union of models we have to use the MERGE operator
*** that takes into account the conflict resolution strategy
*** equivalence relationships, traceability

eq empty-set -> union ( Set ) = Set .
eq Set -> union ( empty-set ) = Set .

eq Set{ N1 } -> union ( Set{ N1 } ) = Set{ N1 } .
eq Set{ N1 } -> union ( Set{ N1 , M22 } ) = Set{ N1, M22 } .
eq Set{ N1 , M11 } -> union ( Set{ N1 } ) = Set{ N1, M11 } .
eq Set{ N1 , M11 } -> union ( Set{ N1 , M22 } ) =
    Set{ N1 } -> includingCol ( Set{ M11 } -> union ( Set{ M22 } ) ) .

eq Set{ M1 } -> union ( Set{ M2 } ) = Set{ M1 , M2 } [owise] .

*** union(bag: Bag(T)): Bag(T)
```

```
eq empty-set -> union ( Bag ) = Bag .
eq Set -> union ( empty-bag ) = Set -> asBag .
eq Set{M1} -> union ( Bag{M2} ) = Bag{M1,M2} [owise] .

*** = (s: Set(T)) : Boolean --> ==

*** intersection(s: Set(T)) : Set(T)
*** intersection(s: Set(T)): Set(T) --> only for sorts that have defined ==
*** to apply the intersection of models we have to use the CROSS operator
*** that takes into account the conflict resolution strategy
*** equivalence relationships, traceability

eq empty-set -> intersection ( Set ) = empty-set .
eq Set -> intersection ( empty-set ) = empty-set .


eq Set{ N1 } -> intersection ( Set{ N1 } ) = Set{ N1 } .
eq Set{ N1, M1 } -> intersection ( Set{ N1 } ) = Set{ N1 } .
eq Set{ N1 } -> intersection ( Set{ N1, M2 } ) = Set{ N1 } .
eq Set{ N1 , M11 } -> intersection ( Set{ N1 , M22 } ) =
    Set{ N1 } -> includingCol ( Set{ M11 } -> intersection ( Set{ M22 } ) ) .
eq Set{ M1 } -> intersection ( Set{ M2 } ) = empty-set [owise] .

*** intersection(bag : Bag(T)): Set(T)
eq empty-set -> intersection ( Bag ) = empty-set .
eq Set -> intersection ( empty-bag ) = empty-set .

eq Set{ N1 } -> intersection ( Bag{ N1 } ) = Set{ N1 } .
eq Set{ N1, M1 } -> intersection ( Bag{ N1 } ) = Set{ N1 } .
eq Set{ N1 } -> intersection ( Bag{ N1, M1 } ) = Set{ N1 } .
eq Set{ N1 , M11 } -> intersection ( Bag{ N1 , M22 } ) =
    Set{ N1 } -> includingCol ( Set{ M11 } -> intersection ( Bag{ M22 } ) ) .

eq Set{ M1 } -> intersection ( Bag{ M2 } ) = empty-set [owise] .

*** difference
*** - (s : Set(T)) : Set(T)
*** - (s: Set(T)): Set(T) --> only for sorts that have defined ==
*** to apply the intersection of models we have to use the DIFF operator
*** that takes into account the conflict resolution strategy
*** equivalence relationships, traceability
op _--_ : Set{T} Set{T} -> Set{T} [prec 33 gather (E e)] .


eq Set{ N1 } -- Set{ N1 } = empty-set .
eq Set{ N1, M1 } -- Set{ N1 } = Set{ M1 } .
eq Set{ N1 } -- Set{ N1, M2 } = empty-set .
eq Set{ N1 , M11 } -- Set{ N1 , M22 } =
    Set{ M11 } --  Set{ M22 }
.
eq Set1 -- Set2 = Set1 [owise] .

eq Set1 -- OSet2 = Set1 -- (OSet2 -> asSet) .

*** including(object: T) : Set(T)
*** including(s : Set(T)) : Set(T)
*** including does not follow the standard. It does not care about
*** uniqueness for the sake of efficiency
eq Set{ M1 } -> including ( M2 ) = Set{ M1 , M2 } .
eq empty-set -> including ( M2 ) = Set{ M2 } .
eq Set{ M1 } -> including ( Col ) = Set{ M1 , Col } .
eq empty-set -> including ( Col ) = Set{ Col } .

*** includingCol: for internal use. it performs the same funcionality
*** that including but takes
*** into account the internal elements of the other collection
eq Set{ M1 } -> includingCol ( Set{ M2 } ) = Set{ M1 , M2 } .
eq empty-set -> includingCol ( Set ) = Set .
eq Set -> includingCol ( empty-set ) = Set .

eq Set -> includingCol ( OSet ) = Set -> includingCol (OSet -> asSet)  .
eq Set -> includingCol ( Bag ) = Set -> includingCol (Bag -> asSet) .
eq Set -> includingCol ( Seq ) = Set -> includingCol (Seq -> asSet) .


*** excluding(object: T): Set(T)
eq Set{N} -> excluding( N ) = empty-set .
eq Set{N,M} -> excluding( N ) = Set{M} .
eq Set -> excluding( N ) = Set [owise] .
```

```
*** symmetricDifference(s: Set(T)): Set(T)
*** XOR semantics


*** flatten(): Set(T2)
op Flatten : Collection{T} Collection{T} -> Collection{T} .

eq Flatten( Set{Col}, Set) = Set -> union (Flatten(Col -> asSet, empty-set)) .
eq Flatten( Set{Col, M1}, Set) =
    Flatten( Set{ M1 }, Set -> union (Flatten(Col -> asSet, empty-set)) ) .
eq Flatten( Set1, Set2) = Set1 -> union(Set2) [owise] .

eq Set -> flatten = Flatten(Set, empty-set) .



*** ****************************************************************
*** Collection{T} iterators
***

*** iterate

eq Set{ N , M } -> iterate ( acc | IF ; Env ; PreConf ) =
    N . IF ( (Set{ M } -> iterate ( acc | IF ; Env ; PreConf )) ; Env ; PreConf ) .
eq Set{ N } -> iterate ( acc | IF ; Env ; PreConf ) =
    N . IF ( acc ; Env ; PreConf ) .
eq empty-set -> iterate ( acc | IF ; Env ; PreConf ) = acc .


*** exists
eq Set{ N , M } -> exists ( BB ; Env ; PreConf ) =
    (N . BB ( Env ; PreConf )) or-else ( Set{ M } -> exists ( BB ; Env ; PreConf )) .
eq Set{ N } -> exists ( BB ; Env ; PreConf ) =
    N . BB ( Env ; PreConf ) .
eq empty-set -> exists ( BB ; Env ; PreConf ) = false .


*** forAll
eq Set{ N , M } -> forAll (  BB ; Env ; PreConf ) =
    (N . BB ( Env ; PreConf )) and-then (Set{ M } -> forAll ( BB ; Env ; PreConf )) .
eq Set{ N } -> forAll ( BB ; Env ; PreConf ) = N . BB ( Env ; PreConf ) .
eq empty-set -> forAll ( BB ; Env ; PreConf ) = true .

*** forAll2
eq empty-set -> forAll2 ( BB ; Env ; PreConf ) = true .
eq Set{ N1 , N2 , M } -> forAll2 ( BB ; Env ; PreConf ) =
    ((N1, N2) . BB ( Env ; PreConf )) and-then
    (Set{ N1 , M } -> forAll2 ( BB ; Env ; PreConf )) and-then
    (Set{ N2 , M } -> forAll2 ( BB ; Env ; PreConf )) .
eq Set{ N1, N2 } -> forAll2 ( BB ; Env ; PreConf ) = (N1, N2) . BB ( Env ; PreConf ) .
eq Set{ N1 } -> forAll2 ( BB ; Env ; PreConf ) = (N1) . BB ( Env ; PreConf ) .

*** isUnique
op different : -> BoolBody{T} .
eq (N1, N2) . different ( Env ; PreConf ) = (N1 =/= N2) .
eq N1 . different ( Env ; PreConf ) = true .

eq Set -> isUnique ( B ; Env ; PreConf ) =
    (Set -> collect ( B ; Env ; PreConf )) -> forAll2 ( different ; Env ; PreConf ) .

*** any
eq Set -> any ( BB ; Env ; PreConf ) =
    ((Set -> select ( BB ; Env ; PreConf )) -> asSequence) -> first  .

*** one
eq Set -> one ( BB ; Env ; PreConf ) =
    ((Set -> select ( BB ; Env ; PreConf )) -> size) == 1  .

*** collect
eq Col -> collect ( B ; Env ; PreConf ) =
    (Col -> collectNested ( B ; Env ; PreConf )) -> flatten .



*** ****************************************************************
*** SET ITERATOR OPERATIONS
***

*** select: Set(T)
op $select : Collection{T} BoolBody{T} Environment PreConfiguration{T}
```

```
    Collection{T} -> Collection{T} .
var ResultCol : Collection{T} .

eq $select( Set{N,M}, BB, Env, PreConf , ResultCol ) =
    if (N . BB ( Env ; PreConf )) then
        $select(Set{ M }, BB, Env, PreConf, ResultCol -> including( N ) )
    else
        $select(Set{ M }, BB, Env, PreConf, ResultCol )
    fi  .

eq $select( Set{N}, BB, Env, PreConf, ResultCol ) =
    if (N . BB ( Env ; PreConf )) then
        ResultCol -> including( N )
    else
        ResultCol
    fi  .

eq $select( empty-set, BB, Env, PreConf, ResultCol ) = ResultCol .

eq Set -> select ( BB ; Env ; PreConf ) =
   $select( Set, BB, Env, PreConf, empty-set) .


*** reject: Set(T)
eq Set{ N , M } -> reject ( BB ; Env ; PreConf ) =
if not(N . BB ( Env ; PreConf )) then
    Set{ N } -> includingCol ( ( Set{ M } -> reject ( BB ; Env ; PreConf )) )
else
    ( Set{ M } -> reject ( BB ; Env ; PreConf ))
fi .

eq Set{ N } -> reject ( BB ; Env ; PreConf ) =
if not(N . BB ( Env ; PreConf )) then
    Set{ N }
else
    empty-set
fi .

eq empty-set -> reject ( BB ; Env ; PreConf ) = empty-set .


*** collectNested: Bag(T)
eq Set{ N , M } -> collectNested ( B ; Env ; PreConf ) =
    Bag{ (N . B ( Env ; PreConf )) }
        -> includingCol ( ( Set{ M } -> collectNested ( B ; Env ; PreConf ) ) ) .
eq Set{ N } -> collectNested ( B ; Env ; PreConf ) = Bag{ (N . B ( Env ; PreConf )) } .
eq empty-set -> collectNested ( B ; Env ; PreConf ) = empty-bag .





*** ************************************************************
*** ************************************************************
*** ************************************************************
*** ************************************************************
***
*** ORDEREDSET ORDERING AUXILIAR OPERATORS
*** Based on Maude list sorting
***

vars A A' L L' : OrderedMagma{T} .


op merge : OrderedSet{T} OrderedSet{T} BoolBody{T} -> OrderedSet{T} .
eq merge (OSet1, OSet2, SortingCriteria) = $merge (OSet1, OSet2, empty-orderedset, SortingCriteria) .

op $merge : OrderedSet{T} OrderedSet{T} OrderedSet{T} BoolBody{T} -> OrderedSet{T} .
eq $merge (OrderedSet{L}, empty-orderedset, OrderedSet{A}, SortingCriteria) = OrderedSet{ A :: L } .
eq $merge (empty-orderedset, OrderedSet{L}, OrderedSet{A}, SortingCriteria) = OrderedSet{A :: L} .

eq $merge (OrderedSet{ E } , OrderedSet{E'}, OrderedSet{A}, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (empty-orderedset, OrderedSet{E'}, OrderedSet{A :: E}, SortingCriteria)
    else
        $merge (OrderedSet{E}, empty-orderedset, OrderedSet{A :: E'}, SortingCriteria)
    fi .
eq $merge (OrderedSet{ E } , OrderedSet{E'}, empty-orderedset, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (empty-orderedset, OrderedSet{E'}, OrderedSet{E}, SortingCriteria)
    else
```

```
        $merge (OrderedSet{E}, empty-orderedset, OrderedSet{E'}, SortingCriteria)
    fi .

eq $merge (OrderedSet{ E } , OrderedSet{E' ::  L'}, OrderedSet{A}, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (empty-orderedset, OrderedSet{E' ::  L'}, OrderedSet{A :: E}, SortingCriteria)
    else
        $merge (OrderedSet{E}, OrderedSet{L'}, OrderedSet{A :: E'}, SortingCriteria)
    fi .
eq $merge (OrderedSet{ E } , OrderedSet{E' ::  L'}, empty-orderedset, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (empty-orderedset, OrderedSet{E' ::  L'}, OrderedSet{E}, SortingCriteria)
    else
        $merge (OrderedSet{E}, OrderedSet{L'}, OrderedSet{E'}, SortingCriteria)
    fi .

eq $merge (OrderedSet{ E :: L} , OrderedSet{E'}, OrderedSet{A}, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (OrderedSet{L}, OrderedSet{E'}, OrderedSet{A :: E}, SortingCriteria)
    else
        $merge (OrderedSet{E :: L}, empty-orderedset, OrderedSet{A :: E'}, SortingCriteria)
    fi .
eq $merge (OrderedSet{ E :: L} , OrderedSet{E' }, empty-orderedset, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (OrderedSet{L}, OrderedSet{E'}, OrderedSet{E}, SortingCriteria)
    else
        $merge (OrderedSet{E :: L}, empty-orderedset, OrderedSet{E'}, SortingCriteria)
    fi .

eq $merge (OrderedSet{ E :: L} , OrderedSet{E' ::  L'}, OrderedSet{A}, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (OrderedSet{L}, OrderedSet{E' ::  L'}, OrderedSet{A :: E}, SortingCriteria)
    else
        $merge (OrderedSet{E :: L}, OrderedSet{L'}, OrderedSet{A :: E'}, SortingCriteria)
    fi .
eq $merge (OrderedSet{ E :: L} , OrderedSet{E' ::  L'}, empty-orderedset, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (OrderedSet{L}, OrderedSet{E' ::  L'}, OrderedSet{E}, SortingCriteria)
    else
        $merge (OrderedSet{E :: L}, OrderedSet{L'}, OrderedSet{E'}, SortingCriteria)
    fi .




sorts $OrderedSplit{T} .
op $orderedsetsplit : OrderedSet{T} OrderedSet{T} OrderedSet{T} -> $OrderedSplit{T} [ctor] .
eq $orderedsetsplit (OrderedSet{E}, OrderedSet{A}, OSet1) =
    $orderedsetsplit (empty-orderedset, OrderedSet{A :: E}, OSet1) .
eq $orderedsetsplit (OrderedSet{E}, empty-orderedset, OSet1) =
    $orderedsetsplit (empty-orderedset, OrderedSet{E}, OSet1) .

eq $orderedsetsplit (OrderedSet{E :: E'}, OrderedSet{A}, OrderedSet{A'}) =
    $orderedsetsplit (empty-orderedset, OrderedSet{A :: E}, OrderedSet{E' :: A'}) .
eq $orderedsetsplit (OrderedSet{E :: E'}, empty-orderedset, OrderedSet{A'}) =
    $orderedsetsplit (empty-orderedset, OrderedSet{E}, OrderedSet{E' :: A'}) .
eq $orderedsetsplit (OrderedSet{E :: E'}, OrderedSet{A}, empty-orderedset) =
    $orderedsetsplit (empty-orderedset, OrderedSet{A :: E}, OrderedSet{E'}) .
eq $orderedsetsplit (OrderedSet{E :: E'}, empty-orderedset, empty-orderedset) =
    $orderedsetsplit (empty-orderedset, OrderedSet{E}, OrderedSet{E'}) .

eq $orderedsetsplit (OrderedSet{E :: L :: E'}, OrderedSet{A}, OrderedSet{A'}) =
    $orderedsetsplit (OrderedSet{L}, OrderedSet{A :: E}, OrderedSet{E' :: A'}) .
eq $orderedsetsplit (OrderedSet{E :: L :: E'}, empty-orderedset, OrderedSet{A'}) =
    $orderedsetsplit (OrderedSet{L}, OrderedSet{E}, OrderedSet{E' :: A'}) .
eq $orderedsetsplit (OrderedSet{E :: L :: E'}, OrderedSet{A}, empty-orderedset) =
    $orderedsetsplit (OrderedSet{L}, OrderedSet{A :: E}, OrderedSet{E'}) .
eq $orderedsetsplit (OrderedSet{E :: L :: E'}, empty-orderedset, empty-orderedset) =
    $orderedsetsplit (OrderedSet{L}, OrderedSet{E}, OrderedSet{E'}) .


op sort : OrderedSet{T} BoolBody{T} -> OrderedSet{T} .
eq sort (empty-orderedset, SortingCriteria) = empty-orderedset .
eq sort (OrderedSet{E}, SortingCriteria) = OrderedSet{E} .
eq sort (OrderedSet{E :: OM}, SortingCriteria) =
    $sort (
        $orderedsetsplit (OrderedSet{E :: OM}, empty-orderedset, empty-orderedset),
        SortingCriteria
        ) .

op $sort : $OrderedSplit{T} BoolBody{T} -> OrderedSet{T} .
```

```
eq $sort ($orderedsetsplit (empty-orderedset, OSet1, OSet2), SortingCriteria) =
    $merge (
        sort (OSet1, SortingCriteria),
        sort (OSet2, SortingCriteria),
        empty-orderedset,
        SortingCriteria
    ) .



*** **********************************************************
*** The operator orderedCollection is not confluent.
*** However, it is only used in the sortedBy operator so that
*** the resulting ordered collection will always be sorted.
***
op orderedCollection : Collection{T} -> Collection{T} .
eq orderedCollection( Set ) = $orderedCollection( Set, empty-orderedset ) .
eq orderedCollection( OSet ) = OSet .
eq orderedCollection( Bag ) = $orderedCollection( Bag, empty-sequence ) .
eq orderedCollection( Seq ) = Seq .



op $orderedCollection : Collection{T} Collection{T} -> Collection{T} .
--- set
eq $orderedCollection( Set{ N1, M1 }, empty-orderedset ) =
    $orderedCollection( Set{ M1 }, OrderedSet{ N1 } ) .
eq $orderedCollection( Set{ N1, M1 }, OrderedSet{ OM } ) =
    $orderedCollection( Set{ M1 }, OrderedSet{ OM :: N1 } ) .
eq $orderedCollection( Set{ N1 }, empty-orderedset ) =
    OrderedSet{ N1 } .
eq $orderedCollection( Set{ N1 }, OrderedSet{ OM } ) =
    OrderedSet{ OM :: N1 } .
--- bag
eq $orderedCollection( Bag{ N1, M1 }, empty-sequence ) =
    $orderedCollection( Bag{ M1 }, Sequence{ N1 } ) .
eq $orderedCollection( Bag{ N1, M1 }, Sequence{ OM } ) =
    $orderedCollection( Bag{ M1 }, Sequence{ OM :: N1 } ) .
eq $orderedCollection( Bag{ N1 }, empty-sequence ) =
    Sequence{ N1 } .
eq $orderedCollection( Bag{ N1 }, Sequence{ OM } ) =
    Sequence{ OM :: N1 } .



--- sortedBy: OrderedSet(T)
--- indicates that the first element is < than the second one
--- eq N1 . MinorOperator ( N2, Env ; PreConf ) =   N1 < N2
eq Set -> sortedBy ( SortingCriteria ; Env ; PreConf ) =
    sort( orderedCollection(Set), SortingCriteria ) .


*** **************************************************************
*** **************************************************************
*** ORDEREDSET
***
*** Duplicates are not allowed: this should be check by the user
*** for the sake of efficiency
***
*** **************************************************************
*** **************************************************************



*** *******************************************************************
*** Collection{T} conversions
***
*** asSet(): Set(T)
op $asSet : Collection{T} Set{T} -> Set{T} .

eq OSet -> asSet =
    $asSet( OSet, empty-set ) .

eq $asSet( OrderedSet{ N :: OM }, empty-set) =
    $asSet( OrderedSet{OM}, Set{N} ) .
eq $asSet( OrderedSet{N}, empty-set ) = Set{N} .
eq $asSet( OrderedSet{N :: OM }, Set{M} ) =
    $asSet( OrderedSet{OM}, Set{ N,M } ) .
eq $asSet( OrderedSet{N}, Set{M} ) = Set{N,M} .
eq $asSet( empty-orderedset, Set) = Set .
```

```
*** asOrderedSet(): OrderedSet(T)
eq OSet -> asOrderedSet = OSet .

*** asBag() : Bag(T)
eq OSet -> asBag = (OSet -> asSet) -> asBag .

*** asSequence(): Sequence(T)
eq OrderedSet{ OM } -> asSequence = Sequence{ OM }  .
eq empty-orderedset -> asSequence = empty-sequence .


*** ********************************************************************
*** Collection{T} OPERATIONS

*** size
eq OrderedSet{ N :: OM } -> size = (OrderedSet{ OM } -> size) + 1 .
eq OrderedSet{ N } -> size = 1 .
eq empty-orderedset -> size = 0 .

*** count
eq OSet -> count ( N2 ) = (OSet -> asSet) -> count (N2) .

*** includes(object: T): Boolean
eq OSet -> includes ( N ) =
    (OSet -> count ( N ) ) > 0 .

*** excludes(object: T): Boolean
eq OSet -> excludes ( N ) =
    (OSet -> count ( N ) ) == 0 .

*** includesAll(c2: Collection{T}(T)): Boolean

*** excludesAll(c2: Collection{T}(T)): Boolean


*** isEmpty(): Boolean
eq empty-orderedset -> isEmpty = true .


*** notEmpty(): Boolean
eq empty-orderedset -> notEmpty = false .

*** sum(): T

*** product(c2: Collection{T}(T2)) : Set(Tuple(first: T, second: T2))

*** ********************************************************************
*** ORDEREDSET OPERATIONS

*** union(s: OrderedSet(T)) :  OrderedSet(T)
eq OSet1 -> union ( OSet2 ) = OSet1 -> appendCol ( OSet2 ) .

eq empty-orderedset -> union ( Seq2 ) = Seq2 .
eq OrderedSet{ OM1 } -> union ( empty-sequence ) = Sequence{ OM1 } .
eq OrderedSet{ OM1 }  -> union ( Sequence{ OM2 } ) = Sequence{ OM1 :: OM2 }  .


*** flatten() : OrderedSet(T2)
*** OrderedSet
eq Flatten( OrderedSet{Col :: OM1}, OSet2) =
    Flatten( OrderedSet{ OM1 },
        (
            OSet2 -> appendCol (
                Flatten(Col -> asOrderedSet, empty-orderedset)
            )
        )
    ) .

eq Flatten( OrderedSet{Col}, OSet2) =
    OSet2 -> appendCol (
        Flatten(Col -> asOrderedSet, empty-orderedset)
    ) .

ceq Flatten( OrderedSet{N :: OM1}, OSet2) =
    Flatten( OrderedSet{ OM1 }, OSet2 -> append( N ) )
if not (N :: Collection{T}) .

ceq Flatten( OrderedSet{N}, OSet2) =
    OSet2 -> append( N )
if not (N :: Collection{T}) .
```

```
eq Flatten( empty-orderedset, Col) = Col .

eq OSet -> flatten = Flatten(OSet, empty-orderedset) .



*** append(object: T): OrderedSet(T)
ceq OrderedSet{ OM } -> append ( N ) = OrderedSet{ OM :: N } if OrderedSet{ OM } -> excludes (N) .
ceq OrderedSet{ OM } -> append ( N ) = OrderedSet{ OM } if OrderedSet{ OM } -> includes (N) .
eq empty-orderedset -> append ( N ) = OrderedSet{ N } .
ceq OrderedSet{ OM } -> append ( Col ) = OrderedSet{ OM :: Col } if OrderedSet{ OM } -> excludes (Col) .
ceq OrderedSet{ OM } -> append ( Col ) = OrderedSet{ OM } if OrderedSet{ OM } -> includes (Col) .
eq empty-orderedset -> append ( Col ) = OrderedSet{ Col } .

*** appendCol(c2: OrderedSet(T1)) : OrderedSet(T1)
eq OrderedSet{ N1 } -> appendCol ( OrderedSet{ N2 } ) = OrderedSet{ N1 } -> append ( N2 ) .
eq OrderedSet{ N1 } -> appendCol ( OrderedSet{ N2 :: OM2 } ) =
    (OrderedSet{ N1 } -> append ( N2 )) -> appendCol( OrderedSet{ OM2 } ) .
eq OrderedSet{ N1 :: OM1 } -> appendCol ( OrderedSet{ N2 } ) = OrderedSet{ N1 :: OM1 } -> append ( N2 ) .
eq OrderedSet{ N1 :: OM1 } -> appendCol ( OrderedSet{ N2 :: OM2 } ) =
    (OrderedSet{ N1 :: OM1 } -> append ( N2 )) -> appendCol ( OrderedSet{ OM2 }) .
eq empty-orderedset -> appendCol ( OrderedSet{ OM2 } ) = OrderedSet{ OM2 } .
eq OrderedSet{ OM1 } -> appendCol ( empty-orderedset ) = OrderedSet{ OM1 } .
eq empty-orderedset -> appendCol ( empty-orderedset ) = empty-orderedset .

*** prepend(object: T): OrderedSet(T)
ceq OrderedSet{ OM } -> prepend ( N ) = OrderedSet{ N :: OM } if OrderedSet{ OM } -> excludes (N) .
ceq OrderedSet{ OM } -> prepend ( N ) = OrderedSet{ OM } if OrderedSet{ OM } -> includes (N) .
eq empty-orderedset -> prepend ( N ) = OrderedSet{ N } .
ceq OrderedSet{ OM } -> prepend ( Col ) = OrderedSet{ Col :: OM } if OrderedSet{ OM } -> excludes (Col) .
ceq OrderedSet{ OM } -> prepend ( Col ) = OrderedSet{ OM } if OrderedSet{ OM } -> includes (Col) .
eq empty-orderedset -> prepend ( Col ) = OrderedSet{ Col } .


*** prependCol(c2: OrderedSet(T1)) : OrderedSet(T1)
eq OrderedSet{ N1 } -> prependCol ( OrderedSet{ N2 } ) = OrderedSet{ N1 } -> prepend ( N2 ) .
eq OrderedSet{ N1 } -> prependCol ( OrderedSet{ N2 :: OM2 } ) =
    (OrderedSet{ N1 } -> prepend ( N2 )) -> prependCol( OrderedSet{ OM2 } ) .
eq OrderedSet{ N1 :: OM1 } -> prependCol ( OrderedSet{ N2 } ) = OrderedSet{ N1 :: OM1 } -> prepend ( N2 ) .
eq OrderedSet{ N1 :: OM1 } -> prependCol ( OrderedSet{ N2 :: OM2 } ) =
    (OrderedSet{ N1 :: OM1 } -> prepend ( N2 )) -> prependCol ( OrderedSet{ OM2 }) .
eq empty-orderedset -> prependCol ( OrderedSet{ OM2 } ) = OrderedSet{ OM2 } .
eq OrderedSet{ OM1 } -> prependCol ( empty-orderedset ) = OrderedSet{ OM1 } .
eq empty-orderedset -> prependCol ( empty-orderedset ) = empty-orderedset .

*** insertAt(index: Integer, object: T) : OrderedSet(T)
ceq OSet -> insertAt ( i ; N ) =
    OSet -> prepend ( N )
if i == 0 .
ceq OrderedSet{ N1 :: OM } -> insertAt ( i ; N2 ) =
    (OrderedSet{ OM } -> insertAt ( (i - 1) ; N2 )) -> prepend ( N1 )
if i > 0 .
ceq OrderedSet{ N1 } -> insertAt ( i ; N2 ) =
    OrderedSet{ N2 } -> prepend ( N1 )
if i > 0 .
eq empty-orderedset -> insertAt ( i ; N ) = OrderedSet{ N } .

*** subOrderedSet(lower: integer, upper : Integer) : OrderedSet(T)

*** at(i: Integer) : T
ceq OrderedSet{ N :: OM } -> at ( i ) =
    N
if i == 0 .
ceq OrderedSet{ N } -> at ( i ) =
    N
if i == 0 .
ceq OrderedSet{ N :: OM } -> at ( i ) =
    OrderedSet{ OM } -> at ( (i - 1) )
if i > 0 .


*** indexOf(obj : T) : Integer
op _->`computeIndexOf`(_;_`) : Collection{T} Int Collection+{T} -> Int .
ceq OrderedSet{ N1 :: OM } -> computeIndexOf ( i ; N2 ) =
    i
if (N1 == N2) .
ceq OrderedSet{ N1 } -> computeIndexOf ( i ; N2 ) =
    i
if (N1 == N2 ) .
ceq OrderedSet{ N1 :: OM } -> computeIndexOf ( i ; N2 ) =
        OrderedSet{ OM } -> computeIndexOf ( (i + 1) ; N2 )
    if (N1 =/= N2) .


    eq OSet -> indexOf ( N2 ) =
        OSet -> computeIndexOf ( 0 ; N2 ) .


    *** first() : T
    eq OrderedSet{ N1 :: OM } -> first = N1 .
    eq OrderedSet{ N1 } -> first = N1 .
--- eq empty-orderedset -> first = empty-orderedset .

    *** last() : T
    eq OrderedSet{ OM :: N1 } -> last = N1 .
    eq OrderedSet{ N1 } -> last = N1 .
--- eq empty-orderedset -> last = empty-orderedset .



    *** difference
    eq OSet1 -- OSet2 = minus(OSet1, OSet2 -> asSet, empty-orderedset) .
    eq OSet1 -- Set2 = minus(OSet1, Set2, empty-orderedset) .

    op minus : Collection{T} Collection{T} Collection{T} -> Collection{T} .
    eq minus(OSet1, empty-set, OSet2) = OSet1 .
    eq minus(empty-orderedset, Set, OSet2) = OSet2 .
    eq minus(OrderedSet{N}, Set{N,M}, OSet) = empty-orderedset .
    eq minus(OrderedSet{N :: OM}, Set{N}, OSet) = OrderedSet{ OM } .
    eq minus(OrderedSet{N :: OM}, Set{N,M}, OSet) =
        minus(OrderedSet{OM}, Set{M}, OSet) .
    eq minus(OSet, Set, OSet2) = OSet2 [owise] .

    *** including(object: T): OrderedSet(T)
    *** does not take uniqueness into account for the sake efficiency,
    *** allowing the insertion of one element when it is known that the element
    *** is not in the orderedset already
    *** eq OSet -> including ( N ) = OSet -> prepend ( N ) .  *** this axiom forces uniqueness

    eq OrderedSet{ OM } -> including ( N ) = OrderedSet{ N :: OM } .
    eq empty-orderedset -> including ( N ) = OrderedSet{ N } .


    *** includingCol: for internal use
    *** does not take uniqueness into account
    eq OrderedSet{ OM1 } -> includingCol ( OrderedSet{ OM2 } ) = OrderedSet{ OM1 :: OM2 } .
    eq empty-orderedset -> includingCol ( OSet ) = OSet .
    eq OSet -> includingCol ( empty-orderedset ) = OSet .

*** eq OSet -> includingCol ( Set ) = OSet -> includingCol (Set -> asOrderedSet) .
*** eq OSet -> includingCol ( Bag ) = OSet -> includingCol (Bag -> asOrderedSet) .
    eq OSet -> includingCol ( Seq ) = OSet -> includingCol (Seq -> asOrderedSet) .


    *** excluding(object: T): OrderedSet(T)
    *** when we exclude one element
    eq OSet -> excluding ( N1 ) =
        excludingHidden(OSet, N1, empty-orderedset) .

    op excludingHidden : Collection{T} Collection+{T} Collection{T} -> Collection{T} .
    eq excludingHidden( OrderedSet{ N1 :: OM1 }, N1, OrderedSet{ OM2 } ) = OrderedSet{ OM2 :: OM1 } .
    eq excludingHidden( OrderedSet{ N1 }, N1, OrderedSet{ OM2 } ) = OrderedSet{ OM2 } .
    eq excludingHidden( OrderedSet{ N1 :: OM1 }, N1, empty-orderedset ) = OrderedSet{ OM1 } .
    eq excludingHidden( OrderedSet{ N1 }, N1, empty-orderedset ) = empty-orderedset .
    eq excludingHidden( OSet1, N2, OSet2 ) = OSet2 -> appendCol ( OSet1 ) [owise] .




    *** ***************************************************************
    *** Collection{T} iterators
    ***

    *** iterate
    eq OrderedSet{ OM :: N } -> iterate ( acc | IF ; Env ; PreConf ) =
        N . IF ( (OrderedSet{ OM } -> iterate ( acc | IF ; Env ; PreConf )) ; Env ; PreConf ) .
    eq OrderedSet{ N } -> iterate ( acc | IF ; Env ; PreConf ) =
        N . IF ( acc ; Env ; PreConf ) .
    eq empty-orderedset -> iterate ( acc | IF ; Env ; PreConf ) = acc .
```

```
*** exists
eq OrderedSet{ N :: OM } -> exists ( BB ; Env ; PreConf ) =
    (N . BB ( Env ; PreConf )) or-else ( OrderedSet{ OM } -> exists ( BB ; Env ; PreConf )) .
eq OrderedSet{ N } -> exists ( BB ; Env ; PreConf ) =
    N . BB ( Env ; PreConf ) .
eq empty-orderedset -> exists ( BB ; Env ; PreConf ) = false .


*** forAll
eq OrderedSet{ N :: OM } -> forAll (  BB ; Env ; PreConf ) =
    (N . BB ( Env ; PreConf )) and-then (OrderedSet{ OM } -> forAll ( BB ; Env ; PreConf )) .
eq OrderedSet{ N } -> forAll ( BB ; Env ; PreConf ) = N . BB ( Env ; PreConf ) .
eq empty-orderedset -> forAll ( BB ; Env ; PreConf ) = true .


*** forAll2
eq OSet -> forAll2 ( BB ; Env ; PreConf ) =
    OSet -> asSet -> forAll2( BB ; Env ; PreConf ) .

*** isUnique
eq OSet -> isUnique ( B ; Env ; PreConf ) =
    (OSet -> collect ( B ; Env ; PreConf )) -> forAll2 ( different ; Env ; PreConf ) .

*** any
eq OSet -> any ( BB ; Env ; PreConf ) =
    (OSet -> select ( BB ; Env ; PreConf )) -> first  .

*** one
eq OSet -> one ( BB ; Env ; PreConf ) =
    ((OSet -> select ( BB ; Env ; PreConf )) -> size) == 1  .




*** ****************************************************************
*** OrderedSet iterators
***

*** select: OrderedSet(T)
*** eq OSet -> select ( BB ; Env ; PreConf ) = OSet -> asSet -> select ( BB ; Env ; PreConf )  .

*** select: OrderedSet(T)
eq OrderedSet{ N :: OM } -> select ( BB ; Env ; PreConf ) =
    if (N . BB ( Env ; PreConf )) then
        (( ( OrderedSet{ OM } -> select ( BB ; Env ; PreConf )) ) -> prepend ( N ) )
    else
        ( OrderedSet{ OM } -> select ( BB ; Env ; PreConf ))
    fi .

eq OrderedSet{ N } -> select ( BB ; Env ; PreConf ) =
    if (N . BB ( Env ; PreConf )) then
        OrderedSet{ N }
    else
        empty-orderedset
    fi .

eq empty-orderedset -> select ( BB ; Env ; PreConf ) = empty-orderedset .




*** reject: OrderedSet(T)
eq OrderedSet{ N :: OM } -> reject ( BB ; Env ; PreConf ) =
    if not (N . BB ( Env ; PreConf )) then
        (( ( OrderedSet{ OM } -> reject ( BB ; Env ; PreConf )) ) -> prepend ( N ) )
    else
        ( OrderedSet{ OM } -> reject ( BB ; Env ; PreConf ))
    fi .

eq OrderedSet{ N } -> reject ( BB ; Env ; PreConf ) =
    if not(N . BB ( Env ; PreConf )) then
        (OrderedSet{ N })
    else
        empty-orderedset
    fi .

eq empty-orderedset -> reject ( BB ; Env ; PreConf ) = empty-orderedset .


*** collectNested: Sequence(T)
eq OrderedSet{ N :: OM } -> collectNested ( B ; Env ; PreConf ) =
    Sequence{ (N . B ( Env ; PreConf )) }
        -> appendCol ( ( OrderedSet{ OM } -> collectNested ( B ; Env ; PreConf ) ) ) .
eq OrderedSet{ N } -> collectNested ( B ; Env ; PreConf ) =
    Sequence{ (N . B ( Env ; PreConf )) } .
eq empty-orderedset -> collectNested ( B ; Env ; PreConf ) = empty-sequence .

*** sortedBy: OrderedSet(T)
eq OSet -> sortedBy ( SortingCriteria ; Env ; PreConf ) =
    sort( OSet, SortingCriteria ) .




*** ****************************************************************
*** ****************************************************************
*** BAG
*** ****************************************************************
*** ****************************************************************



*** *********************************************************************
*** Collection{T} conversions
***
*** asSet(): Set(T)
eq Bag{ N , M } -> asSet = (Bag{ M } -> asSet) -> union ( Set{ N } ) .
eq Bag{ N } -> asSet = Set{ N } .
eq empty-bag -> asSet = empty-set .

*** asOrderedSet(): OrderedSet(T)

*** asBag() : Bag(T)
eq Bag -> asBag = Bag .

*** asSequence(): Sequence(T)


*** *********************************************************************
*** Collection{T} OPERATIONS

*** size
eq Bag{ N , M } -> size = (Bag{ M } -> size) + 1 .
eq Bag{ N } -> size = 1 .
eq empty-bag -> size = 0 .

*** count
eq Bag{ N1 , M } -> count ( N1 ) =
    1 + (Bag{ M } -> count ( N1 )) .
eq Bag{ N1 } -> count ( N1 ) = 1 .
eq empty-bag -> count ( N1 ) = 0 .
eq Bag -> count ( N1 ) = 0 [owise] .

*** includes(object: T): Boolean
eq Bag -> includes ( N ) =
    (Bag -> count ( N ) ) > 0 .
eq empty-bag -> includes ( N ) = false .

*** excludes(object: T): Boolean
eq Bag -> excludes ( N ) =
    (Bag -> count ( N ) ) == 0 .
eq empty-bag -> excludes ( N ) = true .


*** includesAll(c2: Collection{T}(T)): Boolean

*** excludesAll(c2: Collection{T}(T)): Boolean

*** isEmpty(): Boolean
eq empty-bag -> isEmpty = true  .

*** notEmpty(): Boolean
eq empty-bag -> notEmpty = false .

*** sum(): T

*** product(c2: Collection{T}(T2)) : Set(Tuple(first: T, second: T2))


*** *********************************************************************
*** BAG OPERATIONS
***
```

```
*** = (bag : Bag(T)) : Boolean

*** union(bag : Bag(T)) : Bag(T)
eq empty-bag -> union ( Bag ) = Bag .
eq Bag -> union ( empty-bag ) = Bag .
eq Bag{ M1 } -> union ( Bag{ M2 } ) = Bag{ M1 , M2 } .

*** union (set: Set(T)) : Bag(T)
eq empty-bag -> union ( Set ) = Set -> asBag .
eq Bag -> union ( empty-set ) = Bag .
eq Bag{ M1 } -> union ( Set{ M2 } ) = Bag{ M1 , M2 } .


*** intersection(bag: Bag(T)) : Bag(T)
eq empty-bag -> intersection ( Bag ) = empty-bag .
eq Bag -> intersection ( empty-bag ) = empty-bag .

eq Bag{ N1 } -> intersection ( Bag{ N1 } ) = Bag{ N1 } .

eq Bag{ N1, M11 } -> intersection ( Bag{ N1 } ) = Bag{ N1 }  .

eq Bag{ N1 } -> intersection ( Bag{ N2, M22 } ) = Bag{ N1 } .

eq Bag{ N1, M11 } -> intersection ( Bag{ N1, M22 } ) =
    Bag{ N1 } -> includingCol ( Bag{ M11 } -> intersection ( Bag{ M22 } ) ) .

eq Bag1 -> intersection ( Bag2 ) = empty-bag [owise] .

*** intersection(set : Set(T)) : Set(T)
eq empty-bag -> intersection ( Set ) = empty-set .
eq Bag -> intersection ( empty-set ) = empty-set .

eq Bag{ N1 } -> intersection ( Set{ N1 } ) = Set{ N1 } .

eq Bag{ N1, M11 } -> intersection ( Set{ N1 } ) = Set{ N1} .

eq Bag{ N1 } -> intersection ( Set{ N1, M22 } ) = Set{ N1 } .

eq Bag{ N1, M11 } -> intersection ( Set{ N1, M22 } ) =
    Set{ N1 } -> includingCol ( Set{ M11 } -> intersection ( Set{ M22 } ) ) .

eq Bag -> intersection ( Set ) = empty-set [owise] .


*** including(object : T) : Bag(T)
eq Bag{ M1 } -> including ( M2 ) = Bag{ M1 , M2 } .
eq empty-bag -> including ( M2 ) = Bag{ M2 } .
eq Bag{ M1 } -> including ( Col ) = Bag{ M1 , Col } .
eq empty-bag -> including ( Col ) = Bag{ Col } .

*** includingCol
eq Bag{ M1 } -> includingCol ( Bag{ M2 } ) = Bag{ M1 , M2 } .
eq empty-bag -> includingCol ( Bag ) = Bag .
eq Bag -> includingCol ( empty-bag ) = Bag .

eq Bag -> includingCol ( Set ) = Bag -> includingCol (Set -> asBag) .
eq Bag -> includingCol ( OSet ) = Bag -> includingCol (OSet -> asBag) .
eq Bag -> includingCol ( Seq ) = Bag -> includingCol (Seq -> asBag) .



*** excluding(object : T) : Bag(T)
*** when we exclude one element
ceq Bag{ N1 , M } -> excluding ( N1 ) =
    Bag{ M } -> excluding ( N1 )
if not(N1 :: Collection{T}) .

ceq Bag{ N1 } -> excluding ( N1 ) =
    empty-bag
if not(N1 :: Collection{T}) .

eq empty-bag -> excluding ( N2 ) = empty-bag .

ceq Bag -> excluding ( N2 ) = Bag
if not(N2 :: Collection{T}) [owise] .

*** to exclude a bag of elements
eq Bag -> excluding ( Bag{ N , M } ) =
    Bag -> excluding ( N ) -> excluding ( Bag{ M } ) .
```

```
eq Bag -> excluding ( Bag{ N } ) =
    Bag -> excluding ( N )  .

eq empty-bag -> excluding ( Bag ) = empty-bag .
eq Bag -> excluding ( empty-bag ) = Bag .

*** flatten(): Bag(T2)
eq Flatten( Bag{Col, M1}, Bag) =
    Flatten( Bag{ M1 }, Bag -> includingCol (Flatten(Col -> asBag, empty-bag)) ) .

eq Flatten( Bag{Col}, Bag) =
    Bag -> includingCol (Flatten(Col -> asBag, empty-bag)) .

eq Flatten( Bag1, Bag2) = Bag1 -> includingCol(Bag2) [owise] .

eq Bag -> flatten = Flatten(Bag, empty-bag) .



*** ****************************************************************
*** Collection{T} iterators
***

*** iterate
*** Semantics of the operator iterate for a function that manipulates Set(vString)
eq Bag{ N , M } -> iterate ( acc | IF ; Env ; PreConf ) =
    N . IF ( (Bag{ M } -> iterate ( acc | IF ; Env ; PreConf )) ; Env ; PreConf ) .
eq Bag{ N } -> iterate ( acc | IF ; Env ; PreConf ) =
    N . IF ( acc ;  Env ; PreConf ) .
eq empty-bag -> iterate ( acc | IF ; Env ; PreConf ) = acc .


*** exists
eq Bag{ N , M } -> exists ( BB ; Env ; PreConf ) =
    (N . BB ( Env ; PreConf )) or-else ( Bag{ M } -> exists ( BB ; Env ; PreConf )) .
eq Bag{ N } -> exists ( BB ; Env ; PreConf ) =
    N . BB ( Env ; PreConf ) .
eq empty-bag -> exists ( BB ; Env ; PreConf ) = false .

*** forAll
eq Bag{ N , M } -> forAll (  BB ; Env ; PreConf ) =
    (N . BB ( Env ; PreConf )) and-then (Bag{ M } -> forAll ( BB ; Env ; PreConf )) .
eq Bag{ N } -> forAll ( BB ; Env ; PreConf ) = N . BB ( Env ; PreConf ) .
eq empty-bag -> forAll ( BB ; Env ; PreConf ) = true .

*** forAll2
eq Bag{ N1 , N2 , M } -> forAll2 ( BB ; Env ; PreConf ) =
    ((N1 , N2) . BB ( Env ; PreConf )) and-then
    (Bag{ N1 , M } -> forAll2 ( BB ; Env ; PreConf )) and-then
    (Bag{ N2 , M } -> forAll2 ( BB ; Env ; PreConf )) .
eq Bag{ N1, N2 } -> forAll2 ( BB ; Env ; PreConf ) = (N1, N2) . BB ( Env ; PreConf ) .
eq Bag{ N1 } -> forAll2 ( BB ; Env ; PreConf ) = (N1) . BB ( Env ; PreConf ) .
eq empty-bag -> forAll2 ( BB ; Env ; PreConf ) = true .

*** isUnique
eq Bag -> isUnique ( B ; Env ; PreConf ) =
    (Bag -> collect ( B ; Env ; PreConf )) -> forAll2 ( different ; Env ; PreConf )   .

*** any
eq Bag -> any ( BB ; Env ; PreConf ) =
    ((Bag -> select ( BB ; Env ; PreConf )) -> asSequence) -> first  .

*** one
eq Bag -> one ( BB ; Env ; PreConf ) =
    ((Bag -> select ( BB ; Env ; PreConf )) -> size) == 1  .



*** ****************************************************************
*** BAG iterators
***

*** select: Bag(T)
eq Bag{ N , M } -> select ( BB ; Env ; PreConf ) =
    if (N . BB ( Env ; PreConf )) then
        Bag{ N } -> includingCol ( ( Bag{ M } -> select ( BB ; Env ; PreConf )) )
    else
        ( Bag{ M } -> select ( BB ; Env ; PreConf ))
    fi .

eq Bag{ N } -> select ( BB ; Env ; PreConf ) =
    if (N . BB ( Env ; PreConf )) then
        Bag{ N }
    else
        empty-bag
    fi .

eq empty-bag -> select ( BB ; Env ; PreConf ) = empty-bag .




*** reject: Bag(T)
eq Bag{ N , M } -> reject ( BB ; Env ; PreConf ) =
    if not(N . BB ( Env ; PreConf )) then
        Bag{ N } -> includingCol ( ( Bag{ M } -> reject ( BB ; Env ; PreConf )) )
    else
        ( Bag{ M } -> reject ( BB ; Env ; PreConf ))
    fi .

eq Bag{ N } -> reject ( BB ; Env ; PreConf ) =
    if not(N . BB ( Env ; PreConf )) then
        Bag{ N }
    else
        empty-bag
    fi .

eq empty-bag -> reject ( BB ; Env ; PreConf ) = empty-bag .

*** collectNested: Bag(T)
eq Bag{ N , M } -> collectNested ( B ; Env ; PreConf ) =
    Bag{ (N . B ( Env ; PreConf )) }
        -> includingCol ( ( Bag{ M } -> collectNested ( B ; Env ; PreConf ) ) ) .
eq Bag{ N } -> collectNested ( B ; Env ; PreConf ) = Bag{ (N . B ( Env ; PreConf )) } .
eq empty-bag -> collectNested ( B ; Env ; PreConf ) = empty-bag  .








*** ************************************************************
*** ************************************************************
*** ************************************************************
*** ************************************************************
***
*** SEQUENCE ORDERING AUXILIAR OPERATORS
*** Based on Maude list sorting
***


op merge : Sequence{T} Sequence{T} BoolBody{T} -> Sequence{T} .
eq merge (Seq1, Seq2, SortingCriteria) = $merge (Seq1, Seq2, empty-sequence, SortingCriteria) .

op $merge : Sequence{T} Sequence{T} Sequence{T} BoolBody{T} -> Sequence{T} .
eq $merge (Sequence{L}, empty-sequence, Sequence{A}, SortingCriteria) = Sequence{ A :: L } .
eq $merge (empty-sequence, Sequence{L}, Sequence{A}, SortingCriteria) = Sequence{A :: L} .

eq $merge (Sequence{ E } , Sequence{E'}, Sequence{A}, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (empty-sequence, Sequence{E'}, Sequence{A :: E}, SortingCriteria)
    else
        $merge (Sequence{E}, empty-sequence, Sequence{A :: E'}, SortingCriteria)
    fi .
eq $merge (Sequence{ E } , Sequence{E'}, empty-sequence, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (empty-sequence, Sequence{E'}, Sequence{E}, SortingCriteria)
    else
        $merge (Sequence{E}, empty-sequence, Sequence{E'}, SortingCriteria)
    fi .

eq $merge (Sequence{ E } , Sequence{E' ::  L'}, Sequence{A}, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (empty-sequence, Sequence{E' ::  L'}, Sequence{A :: E}, SortingCriteria)
    else
        $merge (Sequence{E}, Sequence{L'}, Sequence{A :: E'}, SortingCriteria)
    fi .
eq $merge (Sequence{ E } , Sequence{E' ::  L'}, empty-sequence, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (empty-sequence, Sequence{E' ::  L'}, Sequence{E}, SortingCriteria)
    else
        $merge (Sequence{E}, Sequence{L'}, Sequence{E'}, SortingCriteria)
    fi .

eq $merge (Sequence{ E :: L} , Sequence{E'}, Sequence{A}, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (Sequence{L}, Sequence{E'}, Sequence{A :: E}, SortingCriteria)
    else
        $merge (Sequence{E :: L}, empty-sequence, Sequence{A :: E'}, SortingCriteria)
    fi .
eq $merge (Sequence{ E :: L} , Sequence{E' }, empty-sequence, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (Sequence{L}, Sequence{E'}, Sequence{E}, SortingCriteria)
    else
        $merge (Sequence{E :: L}, empty-sequence, Sequence{E'}, SortingCriteria)
    fi .




eq $merge (Sequence{ E :: L} , Sequence{E' ::  L'}, Sequence{A}, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (Sequence{L}, Sequence{E' ::  L'}, Sequence{A :: E}, SortingCriteria)
    else
        $merge (Sequence{E :: L}, Sequence{L'}, Sequence{A :: E'}, SortingCriteria)
    fi .
eq $merge (Sequence{ E :: L} , Sequence{E' ::  L'}, empty-sequence, SortingCriteria) =
    if ((E . SortingCriteria ( VariableNameAux = E' ; nonePreConf )) == true) then
        $merge (Sequence{L}, Sequence{E' ::  L'}, Sequence{E}, SortingCriteria)
    else
        $merge (Sequence{E :: L}, Sequence{L'}, Sequence{E'}, SortingCriteria)
    fi .

sorts $SequenceSplit{T} .
op $sequencesplit : Sequence{T} Sequence{T} Sequence{T} -> $SequenceSplit{T} [ctor] .
eq $sequencesplit (Sequence{E}, Sequence{A}, Seq1) =
    $sequencesplit (empty-sequence, Sequence{A :: E}, Seq1) .
eq $sequencesplit (Sequence{E}, empty-sequence, Seq1) =
    $sequencesplit (empty-sequence, Sequence{E}, Seq1) .

eq $sequencesplit (Sequence{E :: E'}, Sequence{A}, Sequence{A'}) =
    $sequencesplit (empty-sequence, Sequence{A :: E}, Sequence{E' :: A'}) .
eq $sequencesplit (Sequence{E :: E'}, empty-sequence, Sequence{A'}) =
    $sequencesplit (empty-sequence, Sequence{E}, Sequence{E' :: A'}) .
eq $sequencesplit (Sequence{E :: E'}, Sequence{A}, empty-sequence) =
    $sequencesplit (empty-sequence, Sequence{A :: E}, Sequence{E'}) .
eq $sequencesplit (Sequence{E :: E'}, empty-sequence, empty-sequence) =
    $sequencesplit (empty-sequence, Sequence{E}, Sequence{E'}) .

eq $sequencesplit (Sequence{E :: L :: E'}, Sequence{A}, Sequence{A'}) =
    $sequencesplit (Sequence{L}, Sequence{A :: E}, Sequence{E' :: A'}) .
eq $sequencesplit (Sequence{E :: L :: E'}, empty-sequence, Sequence{A'}) =
    $sequencesplit (Sequence{L}, Sequence{E}, Sequence{E' :: A'}) .
eq $sequencesplit (Sequence{E :: L :: E'}, Sequence{A}, empty-sequence) =
    $sequencesplit (Sequence{L}, Sequence{A :: E}, Sequence{E'}) .
eq $sequencesplit (Sequence{E :: L :: E'}, empty-sequence, empty-sequence) =
    $sequencesplit (Sequence{L}, Sequence{E}, Sequence{E'}) .


op sort : Sequence{T} BoolBody{T} -> Sequence{T} .
eq sort (empty-sequence, SortingCriteria) = empty-sequence .
eq sort (Sequence{E}, SortingCriteria) = Sequence{E} .
eq sort (Sequence{E :: OM}, SortingCriteria) =
    $sort (
        $sequencesplit (Sequence{E :: OM}, empty-sequence, empty-sequence),
        SortingCriteria
        ) .

op $sort : $SequenceSplit{T} BoolBody{T} -> Sequence{T} .
eq $sort ($sequencesplit (empty-sequence, Seq1, Seq2), SortingCriteria) =
    $merge (
        sort (Seq1, SortingCriteria),
        sort (Seq2, SortingCriteria),
        empty-sequence,
        SortingCriteria
    ) .

*** sortedBy: Sequence(T)
eq Bag -> sortedBy ( SortingCriteria ; Env ; PreConf ) =
    sort( orderedCollection(Bag), SortingCriteria ) .
```

```
*** ******************************************************************
*** ******************************************************************
*** SEQUENCE
*** ******************************************************************
*** ******************************************************************



*** ***********************************************************************
*** Collection{T} conversions
***
*** asSet(): Set(T)
eq Sequence{ N :: OM } -> asSet = (Sequence{ OM } -> asSet) -> union ( Set{ N } ) .
eq Sequence{ N } -> asSet = Set{ N } .
eq empty-sequence -> asSet = empty-set .

*** asOrderedSet(): OrderedSet(T)
eq Sequence{ N :: OM } -> asOrderedSet = ( Sequence{ OM } -> asOrderedSet ) -> prepend ( N ) .
eq Sequence{ N } -> asOrderedSet = OrderedSet{ N } .
eq empty-sequence -> asOrderedSet = empty-orderedset .

*** asBag() : Bag(T)
eq Sequence{ N :: OM } -> asBag = (Sequence{ OM } -> asBag) -> including ( N ) .
eq Sequence{ N } -> asBag = Bag{ N } .
eq empty-sequence -> asBag = empty-bag .

*** asSequence(): Sequence(T)
eq Seq -> asSequence = Seq .




*** ***********************************************************************
*** Collection{T} OPERATIONS

*** size
eq Sequence{ N :: OM } -> size = (Sequence{ OM } -> size) + 1 .
eq Sequence{ N } -> size = 1 .
eq empty-sequence -> size = 0 .


*** count
eq Seq -> count ( N1 ) = (Seq -> asBag) -> count ( N1 ) .

*** includes(object: T): Boolean
eq Seq -> includes ( N ) =
    (Seq -> count ( N ) ) > 0 .
eq empty-sequence -> includes ( N ) = false .

*** excludes(object: T): Boolean
eq Seq -> excludes ( N ) =
    (Seq -> count ( N ) ) == 0 .
eq empty-sequence -> excludes ( N ) = true .

*** includesAll(c2: Collection{T}(T)): Boolean

*** excludesAll(c2: Collection{T}(T)): Boolean



*** isEmpty(): Boolean
eq empty-sequence -> isEmpty = true .

*** notEmpty(): Boolean
eq empty-sequence -> notEmpty = false .

*** sum(): T

*** product(c2: Collection{T}(T2)) : Set(Tuple(first: T, second: T2))


*** ******************************************************************
*** SEQUENCE OPERATIONS
***

*** = (s: Sequence(T)) : Boolean

*** union(s: Sequence(T)) :  Sequence(T)
eq empty-sequence -> union ( Seq2 ) = Seq2  .
eq Seq1 -> union ( empty-sequence ) = Seq1 .
eq Sequence{ OM1 }  -> union ( Sequence{ OM2 } ) = Sequence{ OM1 :: OM2 }  .

eq empty-sequence -> union ( OrderedSet{OM2} ) = Sequence{OM2} .
eq Seq1 -> union ( empty-orderedset ) = Seq1  .
eq Sequence{ OM1 }  -> union ( OrderedSet{ OM2 } ) = Sequence{ OM1 :: OM2 }  .



*** appendCol(c2: Sequence(T1)) : OrderedSet(T1)
eq Sequence{ OM1 } -> appendCol ( Sequence{ OM2 } ) = Sequence{ OM1 :: OM2 } .
eq empty-sequence -> appendCol ( Sequence{ OM2 } ) = Sequence{ OM2 } .
eq Sequence{ OM1 } -> appendCol ( empty-sequence ) = Sequence{ OM1 } .
eq empty-sequence -> appendCol ( empty-sequence ) = empty-sequence .

*** append(object: T): Sequence(T)
eq Sequence{ OM } -> append ( N ) = Sequence{ OM :: N } .
eq empty-sequence -> append ( N ) = Sequence{ N } .
eq Sequence{ OM } -> append ( Col ) = Sequence{ OM :: Col } .
eq empty-sequence -> append ( Col ) = Sequence{ Col } .


*** prependCol(c2: Sequence(T1)) : Sequence(T1)
eq Sequence{ OM1 } -> prependCol ( Sequence{ OM2 } ) = Sequence{ OM2 :: OM1 } .
eq empty-sequence -> prependCol ( Sequence{ OM2 } ) = Sequence{ OM2 } .
eq Sequence{ OM1 } -> prependCol ( empty-sequence ) = Sequence{ OM1 } .
eq empty-sequence -> prependCol ( empty-sequence ) = empty-sequence .

*** prepend(object: T): Sequence(T)
eq Sequence{ OM } -> prepend ( N ) = Sequence{ N :: OM } .
eq empty-sequence -> prepend ( N ) = Sequence{ N } .
eq Sequence{ OM } -> prepend ( Col ) = Sequence{ Col :: OM } .
eq empty-sequence -> prepend ( Col ) = Sequence{ Col } .

*** insertAt(index: Integer, object : T): Sequence(T)
ceq Seq -> insertAt ( i ; N ) =
    Seq -> prepend ( N )
if i == 0 .
ceq Sequence{ N1 :: OM } -> insertAt ( i ; N2 ) =
    (Sequence{ OM } -> insertAt ( (i - 1) ; N2 )) -> prepend ( N1 )
if i > 0 .
ceq Sequence{ N1 } -> insertAt ( i ; N2 ) =
    Sequence{ N2 } -> prepend ( N1 )
if i > 0 .
eq empty-sequence -> insertAt ( i ; N ) = Sequence{ N } .

*** subSequence(lower: Integer, upper: Integer) : Sequence(T)

*** at(i: Integer) : T
ceq Sequence{ N :: OM } -> at ( i ) =
    N
if i == 0 .
ceq Sequence{ N } -> at ( i ) =
    N
if i == 0 .
ceq Sequence{ N :: OM } -> at ( i ) =
    Sequence{ OM } -> at ( (i - 1) )
if i > 0 .

*** indexOf(obj : T) : Integer

ceq Sequence{ N1 :: OM } -> computeIndexOf ( i ; N2 ) =
    i
if (N1 == N2) .
ceq Sequence{ N1 } -> computeIndexOf ( i ; N2 ) =
    i
if (N1 == N2) .
ceq Sequence{ N1 :: OM } -> computeIndexOf ( i ; N2 ) =
    Sequence{ OM } -> computeIndexOf ( (i + 1) ; N2 )
if (N1 =/= N2) .


eq Seq -> indexOf ( N2 ) =
    Seq -> computeIndexOf ( 0 ; N2 ) .


*** first() : T
eq Sequence{ N1 :: OM } -> first = N1 .
eq Sequence{ N1 } -> first = N1 .
```

```
*** last() : T
eq Sequence{ OM :: N1 } -> last = N1 .
eq Sequence{ N1 } -> last = N1 .

*** including(object: T): Sequence(T)
eq Seq -> including ( N ) = Seq -> append ( N ) .

*** includingCol: for internal use
eq Sequence{ OM1 } -> includingCol ( Sequence{ OM2 } ) = Sequence{ OM1 :: OM2 } .
eq empty-sequence -> includingCol ( Seq ) = Seq .
eq Seq -> includingCol ( empty-sequence ) = Seq .

eq Seq -> includingCol ( OSet ) = Seq -> includingCol (OSet -> asSequence) .



*** excluding(object: T): Sequence(T)
*** when we exclude one element
ceq Sequence{ N1 :: OM } -> excluding ( N1 ) =
    Sequence{ OM } -> excluding ( N1 )
if not(N1 :: Collection{T}) .

ceq Sequence{ N1 } -> excluding ( N1 ) =
    empty-sequence
if not(N1 :: Collection{T}) .

eq empty-sequence -> excluding ( N2 ) = empty-sequence .

ceq Seq -> excluding (N2) = Seq
if not(N2 :: Collection{T}) [owise] .


*** to exclude a sequence of elements
eq Seq -> excluding ( Sequence{ N :: OM } ) =
    Seq -> excluding ( N ) -> excluding ( Sequence{ OM } ) .

eq Seq -> excluding ( Sequence{ N } ) =
    Seq -> excluding ( N ) .

eq empty-sequence -> excluding ( Seq ) = empty-sequence .
eq Seq -> excluding ( empty-sequence ) = Seq .



*** flatten() : Sequence(T2)
*** Sequence
eq Flatten( Sequence{Col :: OM1}, Seq2) =
    Flatten( Sequence{ OM1 },
        (
            Seq2 -> appendCol (
                Flatten(Col -> asSequence, empty-sequence)
            )
        )
    ) .

eq Flatten( Sequence{Col}, Seq2) =
    Seq2 -> appendCol (
        Flatten(Col -> asSequence, empty-sequence)
    ) .

ceq Flatten( Sequence{N :: OM1}, Seq2) =
    Flatten( Sequence{ OM1 }, Seq2 -> append( N ) )
if not (N :: Collection{T}) .

ceq Flatten( Sequence{N}, Seq2) =
    Seq2 -> append( N )
if not (N :: Collection{T}) .

eq Flatten( empty-sequence, Col) = Col .

eq Seq -> flatten = Flatten(Seq, empty-sequence) .



*** ****************************************************************
*** Collection{T} iterators
***

*** iterate
eq Sequence{ OM :: N } -> iterate ( acc | IF ; Env ; PreConf ) =
    N . IF ( (Sequence{ OM } -> iterate ( acc | IF ; Env ; PreConf )) ; Env ; PreConf ) .
```

```
eq Sequence{ N } -> iterate ( acc | IF ; Env ; PreConf ) =
    N . IF ( acc ; Env ; PreConf ) .
eq empty-sequence -> iterate ( acc | IF ; Env ; PreConf ) = acc .


*** exists
eq Sequence{ N :: OM } -> exists ( BB ; Env ; PreConf ) =
    (N . BB ( Env ; PreConf )) or-else ( Sequence{ OM } -> exists ( BB ; Env ; PreConf )) .
eq Sequence{ N } -> exists ( BB ; Env ; PreConf ) =
    N . BB ( Env ; PreConf ) .
eq empty-sequence -> exists ( BB ; Env ; PreConf ) = false .


*** forAll
eq Sequence{ N :: OM } -> forAll (  BB ; Env ; PreConf ) =
    (N . BB ( Env ; PreConf )) and-then (Sequence{ OM } -> forAll ( BB ; Env ; PreConf )) .
eq Sequence{ N } -> forAll ( BB ; Env ; PreConf ) = N . BB ( Env ; PreConf ) .
eq empty-sequence -> forAll ( BB ; Env ; PreConf ) = true .

*** forAll2
eq Seq -> forAll2 ( BB ; Env ; PreConf ) =
    Seq -> asBag -> forAll2( BB ; Env ; PreConf ) .


*** isUnique
eq Seq -> isUnique ( B ; Env ; PreConf ) =
    (Seq -> collect ( B ; Env ; PreConf )) -> forAll2 ( different ; Env ; PreConf )  .

*** any
eq Seq -> any ( BB ; Env ; PreConf ) =
    (Seq -> select ( BB ; Env ; PreConf )) -> first  .

*** one
eq Seq -> one ( BB ; Env ; PreConf ) =
    ((Seq -> select ( BB ; Env ; PreConf )) -> size) == 1  .


*** collect


*** ****************************************************************
*** Sequence iterators
***

*** select: Sequence(T)
eq Sequence{ N :: OM } -> select ( BB ; Env ; PreConf ) =
    if (N . BB ( Env ; PreConf )) then
        (( ( Sequence{ OM } -> select ( BB ; Env ; PreConf )) ) -> prepend ( N ) )
    else
        ( Sequence{ OM } -> select ( BB ; Env ; PreConf ))
    fi .

eq Sequence{ N } -> select ( BB ; Env ; PreConf ) =
    if (N . BB ( Env ; PreConf )) then
        Sequence{ N }
    else
        empty-sequence
    fi .

eq empty-sequence -> select ( BB ; Env ; PreConf ) = empty-sequence .



*** reject: Sequence(T)
eq Sequence{ N :: OM } -> reject ( BB ; Env ; PreConf ) =
    if not(N . BB ( Env ; PreConf )) then
        (( ( Sequence{ OM } -> reject ( BB ; Env ; PreConf )) ) -> prepend ( N ) )
    else
        ( Sequence{ OM } -> reject ( BB ; Env ; PreConf ))
    fi .

eq Sequence{ N } -> reject ( BB ; Env ; PreConf ) =
    if not(N . BB ( Env ; PreConf )) then
        (Sequence{ N })
    else
        empty-sequence
    fi .

eq empty-sequence -> reject ( BB ; Env ; PreConf ) = empty-sequence .

*** collectNested: Sequence(T)
```

```
    eq Sequence{ N :: OM } -> collectNested ( B ; Env ; PreConf ) =
        Sequence{ (N . B ( Env ; PreConf )) }
            -> appendCol ( ( Sequence{ OM } -> collectNested ( B ; Env ; PreConf ) ) ) .
    eq Sequence{ N } -> collectNested ( B ; Env ; PreConf ) = Sequence{ (N . B ( Env ; PreConf )) } .
    eq empty-sequence -> collectNested ( B ; Env ; PreConf ) = empty-sequence .

    *** sortedBy: Sequence(T) --> inherited
    eq Seq -> sortedBy ( SortingCriteria ; Env ; PreConf ) =
        sort(Seq, SortingCriteria ) .



    *** ******************************************************************
    *** ******************************************************************
    *** OPERATIONS DEFINED FOR ALL COLLECTIONS
    *** ******************************************************************
    *** ******************************************************************

    *** includesAll(c2: Collection{T}(T)): Boolean
    eq Col -> includesAll(empty-set) = true .
    eq Col -> includesAll(Set{N2,M2}) =
        Col -> includes(N2) and-then Col -> includesAll( Set{ M2 } ) .
    eq Col -> includesAll(Set{N2}) = Col -> includes(N2) .

    eq Col -> includesAll(empty-orderedset) = true .
    eq Col -> includesAll(OrderedSet{N2 :: OM2}) =
        Col -> includes(N2) and-then Col -> includesAll( OrderedSet{ OM2 } ) .
    eq Col -> includesAll(OrderedSet{N2}) = Col -> includes(N2) .

    eq Col -> includesAll(empty-bag) = true .
    eq Col -> includesAll(Bag{N2,M2}) =
        Col -> includes(N2) and-then Col -> includesAll( Bag{ M2 } ) .
    eq Col -> includesAll(Bag{N2}) = Col -> includes(N2) .

    eq Col -> includesAll(empty-sequence) = true .
    eq Col -> includesAll(Sequence{N2 :: OM2}) =
        Col -> includes(N2) and-then Col -> includesAll( Sequence{ OM2 } ) .
    eq Col -> includesAll(Sequence{N2}) = Col -> includes(N2) .


    *** excludesAll(c2: Collection{T}(T)): Boolean
    eq Col -> excludesAll(empty-set) = false .
    eq Col -> excludesAll(Set{N2,M2}) =
        Col -> excludes(N2) and-then Col -> excludesAll( Set{ M2 } ) .
    eq Col -> excludesAll(Set{N2}) = Col -> excludes(N2) .

    eq Col -> excludesAll(empty-orderedset) = false .
    eq Col -> excludesAll(OrderedSet{N2 :: OM2}) =
        Col -> excludes(N2) and-then Col -> excludesAll( OrderedSet{ OM2 } ) .
    eq Col -> excludesAll(OrderedSet{N2}) = Col -> excludes(N2) .

    eq Col -> excludesAll(empty-bag) = false .
    eq Col -> excludesAll(Bag{N2,M2}) =
        Col -> excludes(N2) and-then Col -> excludesAll( Bag{ M2 } ) .
    eq Col -> excludesAll(Bag{N2}) = Col -> excludes(N2) .

    eq Col -> excludesAll(empty-sequence) = false .
    eq Col -> excludesAll(Sequence{N2 :: OM2}) =
        Col -> excludes(N2) and-then Col -> excludesAll( Sequence{ OM2 } ) .
    eq Col -> excludesAll(Sequence{N2}) = Col -> excludes(N2) .

endfm
```