Metamodeling Facilities

Kenneth Bacławski* kenb@ccs.neu.edu
Mieczysław Kokar† kokar@coe.neu.edu
Jeffrey Smith‡ jesmith@mc.com

Abstract

The Meta Object Facility (MOF) standard is used to "define and manipulate interoperability between metamodels and their corresponding models." A requirement of the next MOF generation is to provide support for mapping between models using a more general purpose framework for transformations between metamodels. Our initial contribution was to formally describe this framework to show consistency of these model transformations. Additionally, our hope that this formal framework and transformation process will serve as a foundation that may be used for the UML Infrastructure to provide horizontal transformations between families of UML languages. Our goal of manageably achieving a mathematical formalization for the MOF is accomplished by (1) formalizing a simpler Core Modeling Facility (CMF) and (2) specifying the MOF within the CMF. Our intent is to formalize the MOF, not to replace it. Sequels to this paper will add the necessary detail to this math framework such that consistency checking implementations of model transformation can be demonstrated.

Keywords: formal methods, modeling language.

1 Introduction

A *modeling language* is a means of expressing models of data and behavior. A modeling language is typically introduced to fulfill a particular purpose, and as a result, a great variety of modeling languages have been developed. Examples include the Unified Modeling Language (UML) [1], the Common Warehouse Metamodel (CWM) [2], the Interface Definition Language (IDL) of the Common Object Request Broker Architecture (CORBA) [3] and SQL, the standard relational database modeling and query language. Because of this diversity, it is useful to have a single framework that can express every modeling language. A framework that can do this is called a *metamodeling facility.* A metamodeling facility is a four-layer framework as shown in Figure 1.

*College of Computer Science, Northeastern University, Boston, Massachusetts 02115
†Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115
‡Mercury Computer Systems Inc.

Layer	Activity supported	Examples
M3: Meta-Metamodel	Metamodeling. Specifying Modeling languages.	MOF
M2: Metamodel	Modeling. Specifying models.	UML, CWM, SQL, IDL, MOF and CMF
M1: Model	Storing objects and other data.	Particular models, schemas and interface definitions
M0: System State	General applications.	Particular instances of classes, tuples of tables and remotely accessible objects

Figure 1: Metamodeling Facility Layers

There is now a standard metamodeling facility called the Meta Object Facility (MOF) [4]. The MOF itself (as well as any other metamodeling facility) may be regarded as a modeling language as well as a metamodeling facility. MOF is a specialized kind of modeling language. The models of MOF are modeling languages such as UML and SQL. This allows MOF to be expressed in its own terms as well as to map it to other modeling languages. However, the MOF does have a privileged role within the overall framework for it is the singular modeling language at the meta-metamodeling layer.

The expression of one language within another will be called a *description*. Thus UML, SQL and MOF can be described in MOF, and any of these can be described in UML. When a language is described in itself, then the language is said to be *self-described*. Self-description is the basis for *reflection*, the ability of a the constructs of a language to refer their own structure.

In this paper, we introduce a mathematical framework for metamodeling facilities. The two fundamental features of a metamodeling facility are: (1) It has the four layers shown in Figure 1, and (2) The metamodeling facility is self-describing. Our goal is to achieve a mathematical formalization for the MOF. Because of the size and complexity of the full MOF, we use a "bootstrap" technique in which a simpler Core Metamodeling Facility (CMF) is introduced and the full MOF is specified within the core facility. This prevents the mathematics from becoming unwieldy while still achieving the goal of a mathematical formalization of the MOF.

We begin by discussing the mathematical background required in our treatment (section 2). Each layer of the CMF, as shown in Figure 1, is then introduced and discussed in its own section: metamodels in section 4, models in section 5, and system states in section 6. The CMF is then used as the mathematical framework for describing the MOF in section 7.

Subscripts will be used to denote the layer at which a concept is being interpreted. If a concept appears without a subscript, then that concept only occurs on one of the layers. Superscripts will be used to distinguish a particular example of a concept.

2 Prerequisites

We assume a knowledge of the basic mathematical structures such as product and sum (disjoint union) of sets, and the product and sum of functions. The set of boolean values $\{true, false\}$ is denoted **Boolean**. We abuse notation somewhat and regard each summand S_i of a direct sum (disjoint union) $S_1 + S_2 + \cdots + S_n$, as being a subset of the direct sum.

We will assume a knowledge of partially ordered sets and order-preserving functions. In addition to the usual notion of a commutative diagram, we will also make use of a somewhat weaker condition. A diagram of sets

$$\begin{array}{ccc} A & \xrightarrow{a} & B \\ {\scriptstyle c}\uparrow & & \uparrow{\scriptstyle b} \\ C & \xrightarrow[d]{} & D \end{array}$$

for which B is a partially ordered set is said to be a *partially (ordered) commutative diagram* when for every $x \in C$, the inequality $a(c(x)) \geq b(d(x))$. When a diagram satisfies this condition we will denote it by:

$$\begin{array}{ccc} A & \xrightarrow{a} & B \\ {\scriptstyle c}\uparrow & \geq & \uparrow{\scriptstyle b} \\ C & \xrightarrow[d]{} & D \end{array}$$

3 Literals

As in most programming languages, a metamodeling facility begins with a collection of *built-in types*, also called *basic sorts* or *literal sorts*. Typical examples of literal sorts include **String**, **Boolean**, **Integer** and **Real**. Each literal sort has operations, and it may be given an interpretation as a set together with functions that instantiate the operations. The elements of the interpretation of a literal sort are called *literals*. One can specify that one literal sort is a specialization of another, such as **Integer** being a specialization of **Real**. The specialization relationships give the set of literal sorts the structure of a partially ordered set. If a literal sort is a specialization of another, then the interpretation of the former must be contained in the interpretation of the latter.

It is convenient to combine the literal sorts with their interpretations literals in a single partially ordered set in which a literal lies below a literal sort if it is in the interpretation of the literal sort. We formalize this as follows:

Definition 3.1 *A* literal type structure *is an ordered pair* $(Literal, l)$ *such that*

1. *Literal is a partially ordered set.*
2. $l\colon Literal \to \{\mathbf{s}, \mathbf{v}\}$ *is an order-preserving function to the two-element partially ordered set* $\{\mathbf{s}, \mathbf{v}\}$ *in which* $\mathbf{s} > \mathbf{v}$.

3. *For every $x \in \mathit{Literal}$ such that $l(x) = \mathbf{v}$, there exists $y \in \mathit{Literal}$ such that $y > x$ and $l(y) = \mathbf{s}$.*

We will write $\mathit{Literal}_s$ for $l^{-1}(\mathbf{s})$. The elements of $\mathit{Literal}_s$ are the literal sorts of the literal structure. The literals form the set $\mathit{Literal}_v = l^{-1}(\mathbf{v})$. We write $\mathit{type} \subseteq \mathit{Literal}_v \times \mathit{Literal}_s$ for the binary relation $<$, i.e., $\mathit{type}(x, y)$ if and only if $x < y$. The set of literals in the interpretation of $y \in \mathit{Literal}_s$ will be written $I(y)$. The interpretation sets $I(y)$ are partially ordered sets, and their union is the partially ordered set of all values, i.e., $\bigcup_{y \in \mathit{Literal}_s} I(y) = \mathit{Literal}_v$. In most cases, the partially ordered set $I(y)$ will have no nontrivial order relations. We will generally write *Literal* when referring to a literal type structure, and the order-preserving function l will be implicit.

4 Metamodels

In this section we define the concept of a metamodel in the CMF. These are the entities that form the M2 Layer, so we will write $\mathbf{M}_2$ for the collection of all metamodels.

Definition 4.1 *A* metamodel *is an ordered 5-tuple*

$$\Omega = (\mathit{Element}_2, \mathit{Literal}, \mathit{Property}_2, \mathit{domain}_2, \mathit{range}_2)$$

such that:

1. *$\mathit{Element}_2$ and $\mathit{Property}_2$ are partially ordered sets,*

2. *Literal is a literal type structure,*

3. *domain_2 is an order-preserving function $\mathit{domain}_2 \colon \mathit{Property}_2 \to \mathit{Element}_2$, and*

4. *range_2 is an order-preserving function $\mathit{range}_2 \colon \mathit{Property}_2 \to \mathit{Element}_2 + \mathit{Literal}$.*

The elements of $\mathit{Element}_2$ are called *metamodel sorts* or *metasorts* for short. In the UML Specification [1], a metasort is called a *meta model element.* Typical examples of metasorts are **Class**, **Method**, **Package**, etc. The elements of *Literal* are the literal sorts and literals of the metamodel.

The elements of $\mathit{Property}_2$ are the *property sorts.* Each property sort has a domain and a range, given by the functions domain_2 and range_2, respectively. The domain of a property sort is always a metamodel sort, while the range can be either a metamodel sort or a literal sort. If the range of a property sort is a metamodel sort, then the property sort is called a *meta-association.* If the range of a property sort is a literal sort, then the property sort is called a *meta-attribute.* For example, to specify that each class has a name, one introduces an element **name** $\in \mathit{Property}_2$ such that $\mathit{domain}_2(\textbf{name})$ is **Class** and $\mathit{range}_2(\textbf{name})$ is **String**. The meta-attributes are of two kinds depending on whether the value is in $\mathit{Literal}_s$ or $\mathit{Literal}_v$. In the former case, the meta-attribute specifies a “template” that will be instantiated (or

restricted) at the M1 layer. In the latter case, the meta-attribute is already instantiated, and it can only be copied to the M1 layer.

Another notation that is popular for functions is the "dot" notation. Instead of writing $domain_2$(**name**) as above, one can write it **name**.`domain2`. To avoid confusion we will use only the functional notation in this paper.

The partial orders on $Element_2$ and $Property_2$ determine *specialization* and *generalization* relationships. For example, if $t \leq u$ in $Property_2$, then we say that t is a *sub-property sort* of u. The requirement that $domain_2$ and $range_2$ be order-preserving ensures that the domain and range of the property sort t be specializations of the domain and range, respectively, of the property sort u.

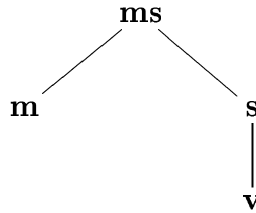
As mentioned in the Introduction above, metamodeling facilities can be regarded as specialized modeling languages (i.e., as a metamodel). Making this precise is a key step in the process of bootstrapping from the CMF to the full MOF. We now define the CMF and the MOF as metamodels.

Example 4.2 The CMF is can be described within itself as the metamodel

$$CMF = (Element_2^C, Literal^C, Property_2^C, domain_2^C, range_2^C)$$

for which:

1. $Element_2^C$ is the 4-element set $\{\mathbf{m}, \mathbf{s}, \mathbf{ms}, \mathbf{v}\}$ with the partial order as shown in the following diagram:



2. $Literal^C$ is the empty set.

3. $Property_2^C$ is the 1-element set $\{\mathbf{p}\}$ The significance of the elements in $Element_2^C$ and $Property_2^C$ will be made clear in Definition 5.2. Their names were inspired as follows:

m	*meta model element*	**s**	*literal sort*
p	*property sort*	**v**	*literal value*

4. $domain_2^C(\mathbf{p}) = \mathbf{m}$ and $range_2^C(\mathbf{p}) = \mathbf{ms}$.

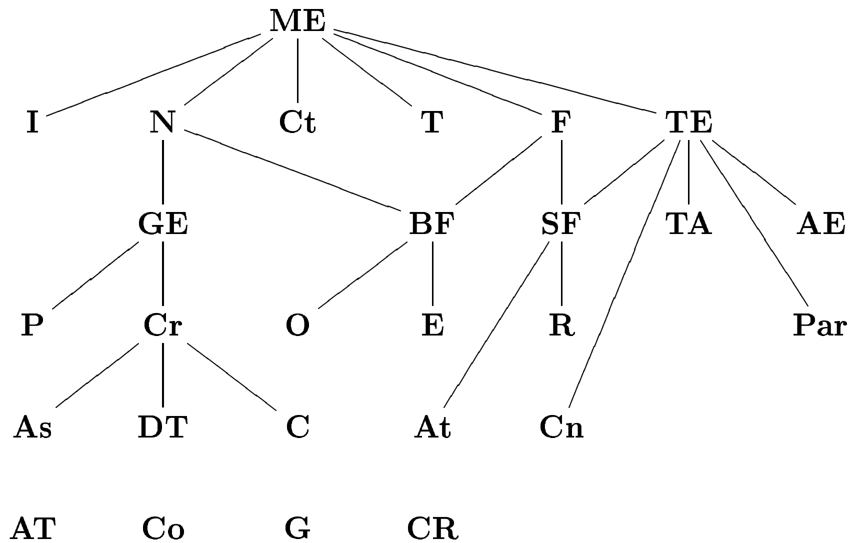
We now describe the MOF as a metamodel which will be the basis for formalizing the MOF framework.

Example 4.3 The Meta Object Facility can be described within the CMF as the metamodel

$$MOF = (Element_2^M, Literal^M, Property_2^M, domain_2^M, range_2^M)$$

for which

1. $Element_2^M$ is the following 27-element partially ordered set:

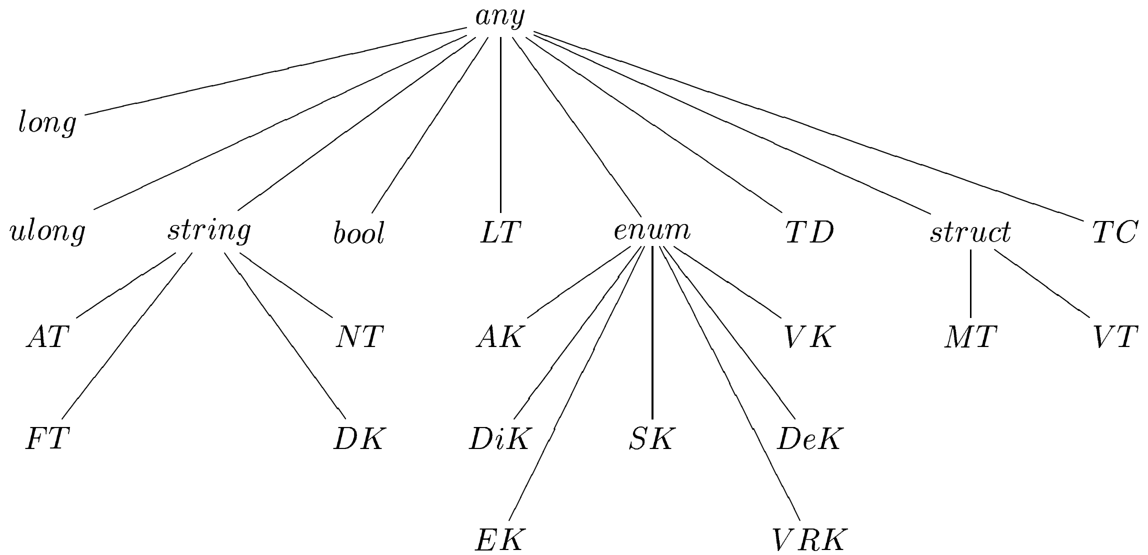


The full names of the elements of $Element_2^M$ are in the following table:

As	Association	**GE**	GeneralizableElement
AE	AssociationEnd	**G**	Generalizes
AT	AttachesTo	**I**	Import
At	Attribute	**ME**	ModelElement
BF	BehavioralFeature	**N**	Namespace
CR	CanRaise	**O**	Operation
C	Class	**P**	Package
Cr	Classifier	**Par**	Parameter
Cn	Constant	**R**	Reference
Ct	Constraint	**SF**	StructuralFeature
Co	Contains	**T**	Tag
DT	DataType	**TA**	TypeAlias
E	Exception	**TE**	TypedElement
F	Feature		

MOF Meta-Metamodel Elements

2. $Literal_s^M$ is the 23-element partially ordered set shown below:



The *enum* and *struct* sorts are not part of the MOF. They were introduced by Richters and Gogolla [5] as abstract supersorts to simplify the definition of operations on MOF literal sorts. The full names of the elements of $Literal_s^M$ are given in the following table:

AK	AggregationKind	VRK	VerifyResultKind
AT	AnnotationType	VT	ViolationType
DK	DependencyKind	VK	VisibilityKind
DeK	DepthKind	TC	TypeCode
DiK	DirectionKind	any	any
EK	EvaluationKind	long	(signed) long
FT	FormatType	ulong	unsigned long
LT	LiteralType	enum	enumeration
MT	MultiplicityType	string	**String**
NT	NameType	struct	structure
SK	ScopeKind	bool	**Boolean**
TD	TypeDescriptor		

MOF Literal Types

3. $Property_2^M$ consists of 35 elements. Of these, 16 are meta-associations, and 29 are meta-attributes. Derived meta-associations (Exposes (**3.5.4**) and DependsOn (**3.5.9**)) and derived meta-attributes are operations, and therefore not included in this partially ordered set. There are no non-trivial order relations between any of the elements of $Property_2^M$. The set $Property_2^M$ and the values of $domain_2^M$ and $range_2^M$ are defined together in the following tables:

Member of $Property_2^M$	$domain_2^M$	$range_2^M$
aggregation	**AE**	AggregationKind
annotation	**ME**	AnnotationType
direction	**Par**	DirectionKind
evaluationPolicy	**Ct**	EvaluationKind
expression	**Ct**	any
isAbstract	**GE**	**Boolean**
isChangeable_{AE}	**AE**	**Boolean**
isChangeable_{SF}	**SF**	**Boolean**
isClustered	**I**	**Boolean**
isDerived_{As}	**As**	**Boolean**
isDerived_{At}	**At**	**Boolean**
isLeaf	**GE**	**Boolean**
isNavigable	**AE**	**Boolean**
isQuery	**O**	**Boolean**
isRoot	**GE**	**Boolean**
isSingleton	**C**	**Boolean**
language	**Ct**	**String**
multiplicity_{AE}	**AE**	MultiplicityType
$\text{multiplicity}_{Par}$	**Par**	MultiplicityType
multiplicity_{SF}	**SF**	MultiplicityType
name	**ME**	NameType
scope	**F**	ScopeKind
tagID	**T**	**String**
typeCode	**DT**	TypeDescriptor
value	**Cn**	LiteralType
values	**T**	any
visibility_F	**F**	VisibilityKind
visibility_{GE}	**GE**	VisibilityKind
visibility_I	**I**	VisibilityKind

MOF Meta-Attributes

Meta-Associations		
Member of $Property_2^M$	$domain_2^M$	$range_2^M$
aliases	**I**	**N**
constrains	**Ct**	**ME**
containedElement	**Co**	**ME**
container	**Co**	**N**
except	**CR**	**E**
isOfType	**TE**	**Cr**
model	**AT**	**ME**
nextContainedElement	**Co**	**Co**
nextExcept	**CR**	**CR**
nextSupertype	**G**	**G**
nextTag	**AT**	**AT**
operation	**CR**	**O**
refersTo	**R**	**AE**
subtype	**G**	**GE**
supertype	**G**	**GE**
tag	**AT**	**T**

MOF Meta-Assocations

5 Models

The next layer in the metamodeling facility is the Model (M1) Layer. Each model is based on a metamodel. One can think of a model either as an instance of its metamodel or as a specification of instances at the System State (M0) Layer. The collection of all models is written $\mathbf{M}_1$.

Definition 5.1 *A* model Σ based on a metamodel Ω *(or more briefly, a* model Σ of Ω*) is an ordered 6-tuple*

$$(\Omega, Element_1, Property_1, sort_{12}, domain_1, range_1)$$

such that:

1. $\Omega = (Element_2, Literal, Property_2, domain_2, range_2)$ *is a metamodel,*

2. $Element_1$ *and* $Property_1$ *are partially ordered sets,*

3. $sort_{12}\colon Element_1 + Property_1 \to Element_2 + Property_2$ *is order-preserving,*

4. $sort_{12}(Element_1) \subseteq Element_2$ *and* $sort_{12}(Property_1) \subseteq Property_2$*,*

5. $domain_1\colon Property_1 \to Element_1$*,*

6. $range_1 \colon Property_1 \rightarrow Element_1 + Literal$, *and*

7. *(Sort Compatibility Conditions) The following diagrams are partially commutative:*

$$\begin{array}{ccc} Property_2 & \xrightarrow{domain_2} & Element_2 \\ {\scriptstyle sort_{12}}\uparrow & \geq & \uparrow{\scriptstyle sort_{12}} \\ Property_1 & \xrightarrow[domain_1]{} & Element_1 \end{array}$$

$$\begin{array}{ccc} Property_2 & \xrightarrow{range_2} & Element_2 + Literal \\ {\scriptstyle sort_{12}}\uparrow & \geq & \uparrow{\scriptstyle sort_{12}+1_{Literal}} \\ Property_1 & \xrightarrow[range_1]{} & Element_1 + Literal \end{array}$$

The elements of $Element_1$ are called *model sorts*, and the elements of $Property_1$ are called the *model properties.* The function $sort_{12}$ maps a model sort or model property to its corresponding metasort.

For example, a data model for company personnel might have a Person class which is an instance of **Class**. This is expressed by the fact that $sort_{12}$ takes the value **Class** on the Person class.

The functions $domain_1$ and $range_1$ determine two characteristics of a model property in the same way that the functions $domain_2$ and $range_2$ determine two characteristics of a property metasort. For example, to specify that "`Person`" is the name of the Person class, one introduces an element $n \in Property_1$ such that $sort_{12}(n)$ is **name** $\in Property_2$, $domain_1(n)$ is the Person class, and $range_1(n)$ is the string "`Person`". In other words, these three maps determine the triple (**name**, Person, "`Person`"). For this reason, model properties are also called *model triples.*

Each property sort (at any layer of the facility), defines a relation from the instances of the domain to the instances of the range (at the next lower layer of the facility). For a property sort $P \in Property_2$, we will write $I(P)(x,y)$ to mean $\exists p(sort_{12}^{p} \leq P \ \& \ domain_1(p) = x \ \& \ range_1(p) = y)$. We call $I(P)$ the *interpretation* of P as a binary relation. The *transpose* or *inverse* relation is written $I(P)^t$. It is defined by $I(P)^t(x,y) \iff I(P)(y,x)$. Given two property sorts, P and Q, their *composition* is a relation written $I(P;Q)$ and defined by $I(P;Q)(x,y) \iff \exists z(I(P)(x,z) \ \& \ I(Q)(z,y))$.

For a metamodel, there were two kinds of property: meta-association and meta-attribute. For a model, there are three kinds of property depending on whether the value of $range_1$ is in $Element_1$, is a literal sort or is a literal. The first kind of property is called a *model association.* It relates two model sorts. The second kind will be called a *data attribute*, and the third will be called a *model attribute.* The example given above is a model attribute. It gives a feature of the Person class at the model layer only. One might also refer to such a property as a *static attribute* (by analogy with static variables in programming languages).

The second kind of attribute is a template for attributes at the system state layer. For example, to specify that a person has an address, one might model this with a property $a \in \mathit{Property}_1$ whose corresponding "triple" is (**address**, Person, **String**). In practice, it would not be reasonable to be defining an **address** meta-attribute at the metamodel layer. A more reasonable way to deal with this is to define an **Attribute** metamodel sort that has a **datatype** meta-attribute whose range_2 is the most general literal type that is allowed for any attribute. To specify that a person has an address, one introduces an instance *address* of the **Attribute** metamodel sort and an instance a of the **datatype** meta-attribute. The corresponding "triple" interpretation of a would then be $I(a) =$ (**datatype**, *address*, **String**). The *address* model sort is linked to the Person class by an instance of a meta-association linking the **Attribute** metasort to the **Class** metasort.

Note that a meta-attribute can have both model attributes and data attributes as instances. In practice, one would constrain a given meta-attribute to allow only one or the other.

In a graphic user interface, model sorts are typically represented by a graphical shape. The type of graphical shape is often used to suggest the sort of the model element. For example, a simple rectangle would be used for a class, while a rectangle with a tab on top (suggesting a file folder) would be used for a package.

Model properties are the "glue" that links together the otherwise disparate elements of a model. Model properties are not typically shown explicitly in a graphic user interface. Their existence is usually indicated by juxtaposition or proximity. For example, the name of the Person class would be shown by placing the string "`Person`" near the top of the rectangle used for representing the Person class in the data model.

There are two alternative notations that one can use for the function sort_{12}. One of them is the "dot" notation already mentioned. Another one that is frequently employed in the literature on multi-sorted logical systems is the *indexed collection* notation. For example, the function $\mathit{sort}_{12}\colon \mathit{Element}_1 \to \mathit{Element}_2$ defines an indexed collection of sets $\{E_m \mid m \in \mathit{Element}_2\}$, where $E_m = \{e \in \mathit{Element}_1 \mid \mathit{sort}_{12}(e) = m\}$. The function notation was used in this paper in preference to the indexed collection notation in order to make the formulas and constructions simpler to state.

The first Sort Compatibility Conditions in Definition 5.1 for an element $n \in \mathit{Property}_1$ can be stated as follows:

$$\mathit{domain}_2(\mathit{sort}_{12}(n)) \geq \mathit{sort}_{12}(\mathit{domain}_1(n))$$

$$\mathit{range}_2(\mathit{sort}_{12}(n)) \geq (\mathit{sort}_{12} + 1_{\mathit{Literal}})(\mathit{range}_1(n)).$$

The element n links $\mathit{domain}_1(n)$ to $\mathit{range}_1(n)$. This link must conform to the domain and range conditions for the property sort $\mathit{sort}_{12}(n)$. The first inequality requires that $\mathit{domain}_1(n)$ be in the domain of the property sort of n.

The second Sort Compatibility Condition is similar to the first one for model associations and model attributes: $\mathit{range}_1(n)$ must be in the range of the property sort of n or in a subsort of the range. For data attributes, the condition is somewhat different. In this case, the function f is the identity, so it requires that the $\mathit{range}_1(n)$ be *equal to* the range of the

property sort of n or to a subsort of the range, rather than a *member of* as in the case of the other two kinds of property. In effect, a meta-attribute determines the most general literal sort that is allowed, while particular data attributes can restrict to a literal sort that is more specific. At the system state layer, an instance of a data attribute further restricts the possibilities by choosing a specific value.

As an example of the Sort Compatibility Conditions, consider the model property $n \in \mathit{Property}_1$ introduced above, which specifies that the Person class has the name "`Person`". The metasort of this model property is $\mathit{sort}_{12}(n) = \mathbf{name} \in \mathit{Property}_2$, whose domain is **Class** and whose range is **String**. Now $\mathit{domain}_1(n)$ is the Person class, so its metasort is the class metamodel element $\mathbf{Class} \in \mathit{Element}_2$. Similarly, $\mathit{range}_1(n)$ is "`Person`", whose metasort is the **String** literal sort, and the Sort Compatibility Conditions are satisfied in this case.

We now return to Example 4.2 and show that every metamodel is described as a model based on *CMF*.

Definition 5.2 For every metamodel $\Omega = (\mathit{Element}_2, \mathit{Literal}, \mathit{Property}_2, \mathit{domain}_2, \mathit{range}_2)$, one can define a 6-tuple $C_{21}(\Omega) = (\mathit{CMF}, \mathit{Element}_1^C, \mathit{Property}_1^C, \mathit{sort}_{12}^C, \mathit{domain}_1^C, \mathit{range}_1^C)$ as follows:

1. $\mathit{Element}_1^C$ is the disjoint union $\mathit{Element}_2 + \mathit{Literal}$.

2. $\mathit{Property}_1^C = \mathit{Property}_2$.

3. $\mathit{sort}_{12}^C \colon \mathit{Element}_1^C \to \mathit{Element}_2^C$ is defined as follows:

   (a) For $x \in \mathit{Element}_1^C$,

   $$\mathit{sort}_{12}^C(x) = \begin{cases} \mathbf{m}, & \text{if } x \in \mathit{Element}_2\text{; } \\ \mathbf{s}, & \text{if } x \in \mathit{Literal}_s\text{; and} \\ \mathbf{v}, & \text{if } x \in \mathit{Literal}_v. \end{cases}$$

   (b) For $x \in \mathit{Property}_1^C$, $\mathit{sort}_{12}^C(x) = \mathbf{p}$.

4. $\mathit{domain}_1^C = \mathit{domain}_2$ and $\mathit{range}_1^C = \mathit{range}_2$.

Proposition 5.3 *For every metamodel Ω, the 6-tuple $C_{21}(\Omega)$ is a model based on CMF.*

Proof By construction, we only need to show that the Sort Compatibility Conditions hold. Furthermore, by the remarks after the definitions of *CMF* and $C_{21}(\Omega)$, it is only necessary to show that the diagrams

$$\begin{array}{ccc} \mathit{Property}_2^C & \xrightarrow{\mathit{domain}_2^C} & \mathit{Element}_2^C \\ \mathit{sort}_{12}^C \uparrow & \geq & \uparrow \mathit{sort}_{12}^C \\ \mathit{Property}_1^C & \xrightarrow[\mathit{domain}_1^C]{} & \mathit{Element}_1^C \end{array}$$

$$\begin{array}{ccc} Property_2^C & \xrightarrow{range_2^C} & Element_2^C \\ sort_{12}^C \uparrow & \geq & \uparrow sort_{12}^C \\ Property_1^C & \xrightarrow[range_1^C]{} & Element_1^C \end{array}$$

are partially commutative. Let $x \in Property_1^C$.

1. To show that the first diagram is partially commutative, first compute $sort_{12}^C(domain_1^C(x))$. This is easily seen to be $sort_{12}^C(domain_2(x)) = \mathbf{m}$, because $domain_2(x) \in Element_2$. In the other direction, $domain_2^C(sort_{12}^C(x)) = domain_2^C(\mathbf{p}) = \mathbf{m}$, so the first diagram commutes.

2. For the second diagram, first compute $sort_{12}^C(range_1^C(x)) = sort_{12}^C(range_2(x))$. Now $range_2(x) \in Element_2 + Literal$, so $sort_{12}^C(range_2(x))$ is either $\mathbf{m}$, $\mathbf{s}$ or $\mathbf{v}$. In the other direction, $range_2^C(sort_{12}^C(x)) = range_2^C(\mathbf{p}) = \mathbf{ms}$. Since $\mathbf{m}$, $\mathbf{s}$ and $\mathbf{v}$ all lie below $\mathbf{ms}$ in $Element_2^C$, the second diagram is partially commutative.

■

The mapping C_{21} is easily seen to be a one-to-one mapping, so it is an embedding of the M2 layer in the M1 layer. Since it is describing CMF metamodels in terms of CMF models, it is a self-description. The converse of Proposition 5.3 does not hold: there are models of *CMF* that do not correspond to a metamodel. We now give necessary and sufficient conditions for a model of *CMF* to correspond to a metamodel.

Theorem 5.4 Let $\Sigma = (CMF, Element_1, Property_1, sort_{12}, domain_1, range_1)$ be a model based on *CMF*. There exists a metamodel Ω such that $C_{21}(\Omega) = \Sigma$ if and only if the following conditions hold:

1. $\forall x \in Element_1(sort_{12}(x) \neq \mathbf{ms})$.

2. $\forall x \in Element_1(sort_{12}(x) = \mathbf{v} \Rightarrow \exists y \in Element_1(sort_{12}(y) = \mathbf{s} \;\&\; x \leq y))$.

Proof It is straightforward to check that for any metamodel Ω, the model $C_{21}(\Omega)$ satisfies all of the conditions. Conversely, suppose that Σ satisfies all of the conditions. Define sets as follows:

$$\begin{array}{ll} E = {sort_{12}}^{-1}(\mathbf{m}) & S = {sort_{12}}^{-1}(\mathbf{s}) \\ V = {sort_{12}}^{-1}(\mathbf{v}) & P = {sort_{12}}^{-1}(\mathbf{p}) \end{array}$$

By Condition 1, $Element_1 = E \cup S \cup V$. By the Sort Compatibility Conditions for Σ, the following hold:

$$domain_1(P) \subseteq E \quad range_1(P) \subseteq E \cup S \cup V$$

The partially ordered set $L = S \cup V$ defines the literal sorts and literals as in Definition 3.1. Condition 2 implies that every element of V belongs to the interpretation of at least one sort in S.

Putting all of the above together, the 5-tuple $(E, L, P, domain_1, range_1)$ satisfies all of the conditions for a metamodel. It remains to show that $C_{21}(E, L, P, domain_1, range_1) = \Sigma$.

1. $Element_1^C = E + L$, and $Element_1 = E \cup L$. By construction, the sets E and L are disjoint so $Element_1^C = Element_1$.
2. $Property_1^C = Property_1$ by definition.
3. $sort_{12}^C$ and $sort_{12}$ coincide by the construction of the sets E, L, P, and by Condition 1.
4. The functions $domain_1^C$ and $domain_1$ coincide by construction, and the same is true of the functions $range_1^C$ and $range_1$.

■

6 System States

The System State layer is concerned with instances that conform to a specification at the Model layer in almost exactly the same way that a model conforms to a metamodel. The System State layer is normally the lowest layer, and for this reason its instances are generally regarded as being "concrete". However, this is by no means required, and there can be still lower layers than the System State layer if desired. The collection of all system states is written $\mathbf{M}_0$.

Definition 6.1 *Let Σ be a model. A* system state *or* data instance *conforming to Σ (or more briefly, a Σ-state) is an ordered 6-tuple*

$$(\Sigma, Element_0, Property_0, sort_{01}, domain_0, range_0)$$

such that:

1. *Σ is a model,*
2. *$Element_0$ and $Property_0$ are partially ordered sets,*
3. *$sort_{01}\colon Element_0 + Property_0 \to Element_1 + Property_1$ is order-preserving,*
4. *$sort_{01}(Element_0) \subseteq Element_1$ and $sort_{01}(Property_0) \subseteq Property_1$,*
5. *$domain_0\colon Property_0 \to Element_0$,*
6. *$range_0\colon Property_0 \to Element_0 + Literal$, and*

7. *(Sort Compatibility Conditions) The following diagrams are partially commutative:*

$$\begin{array}{ccc} Property_1 & \xrightarrow{domain_1} & Element_1 \\ sort_{01}\uparrow & \geq & \uparrow sort_{01} \\ Property_0 & \xrightarrow[domain_0]{} & Element_0 \end{array}$$

$$\begin{array}{ccc} Property_1 & \xrightarrow{range_1} & Element_1 + Literal \\ sort_{01}\uparrow & \geq & \uparrow sort_{01}+1_{Literal} \\ Property_0 & \xrightarrow[range_0]{} & Element_0 + Literal \end{array}$$

Note that the form of a system state is identical to that of a model. This is deliberate. The intention is to allow one to define as many layers in the metamodeling facility as one needs.

The Sort Compatibility Condition above has an interpretation that is essentially identical to that for Definition 5.1, and the relationship between the M1 and M0 layers is almost identical to that between the M2 and M1 layers.

One complication that is not shown in this definition (nor in the definition of a model) is the possibility that links (and model properties) could map to elements on a different layer. Although there is no provision for such links and model properties in the CMF, one can introduce them by reifying a higher layer at a lower layer. The mapping C_{21} in Definition 5.2 is an example of such a reification.

The elements of $Element_0$ are called *objects* or *instances*, and the elements of $Property_0$ are called *links*. The function $sort_{01}$ maps an object or link to its corresponding *model sort* or *model property*. The functions $domain_0$ and $range_0$ determine the characteristics of a link exactly as on the M1 layer.

We now continue Example 4.2, showing that one can describe every model as a system state. The construction of this system state is given in Definition 6.2, and in Theorem 6.4 we give necessary and sufficient conditions for a system state to correspond to a model.

Definition 6.2 For every model $\Sigma = (\Omega, Element_1, Property_1, sort_{12}, domain_1, range_1)$, one can define a 6-tuple $C_{10}(\Sigma) = (C_{21}(\Omega), Element_0^C, Property_0^C, sort_{01}^C, domain_0^C, range_0^C)$ as follows:

1. $Element_0^C$ is the disjoint union $Element_1 + Literal$.

2. $Property_0^C$ is the disjoint union $Property_1$.

3. $sort_{01}^C\colon Element_0^C \to Element_1^C$ is defined as follows:

   (a) For $x \in Element_0^C$,

$$sort_{01}^C(x) = \begin{cases} sort_{12}(x), & \text{if } x \in Element_1\text{; and} \\ x, & \text{if } x \in Literal. \end{cases}$$

(b) For $x \in Property_0^C$, $sort_{01}^C(x) = sort_{12}(x)$.

4. $domain_0^C = domain_1$ and $range_0^C = range_1$.

Note that $Literal^C$ is empty so that $range_0^C \colon Property_0^C \to Element_0^C$.

The definition of $C_{10}(\Sigma)$ is deliberately very similar to $C_{21}(\Omega)$.

Proposition 6.3 *For every model Σ, the 6-tuple $C_{10}(\Sigma)$ is a system state based on $C_{21}(\Omega)$.*

Proof As in the proof of Proposition 5.3, it is only necessary to show that the diagrams

$$\begin{array}{ccc} Property_1^C & \xrightarrow{domain_1^C} & Element_1^C \\ {\scriptstyle sort_{01}^C}\uparrow & \geq & \uparrow{\scriptstyle sort_{01}^C} \\ Property_0^C & \xrightarrow[domain_0^C]{} & Element_0^C \end{array}$$

$$\begin{array}{ccc} Property_1^C & \xrightarrow{range_1^C} & Element_1^C \\ {\scriptstyle sort_{01}^C}\uparrow & \geq & \uparrow{\scriptstyle sort_{01}^C} \\ Property_0^C & \xrightarrow[range_0^C]{} & Element_0^C \end{array}$$

are partially commutative. Let $x \in Property_0^C$.

1. In the first diagram, first compute $sort_{01}^C(domain_0^C(x)) = sort_{01}^C(domain_1(x))$. Since $domain_1(x) \in Element_2$, this is the same as $sort_{12}(domain_1(x))$. In the other direction, $domain_1^C(sort_{01}^C(x)) = domain_1^C(sort_{12}(x))$. Since $sort_{12}(x) \in Property_2$, this is the same as $domain_2(sort_{12}(x))$. So the first diagram partially commutes because of the first Sort Compatibility Condition for Σ.

2. Similarly, the second diagram is partially commutative because of the second Sort Compatibility Condition for Σ.

■

As in the case of Proposition 5.3, there are system states of $C_{21}(\Omega)$ that do not correspond to a model. We now give necessary and sufficient conditions for a system state of $C_{21}(\Omega)$ to correspond to a model.

Theorem 6.4 Let Ω be a metamodel, and let

$$\Delta = (C_{21}(\Omega), Element_0, Property_0, sort_{01}, domain_0, range_0)$$

be a system state based on $C_{21}(\Omega)$. There exists a model Σ such that $C_{10}(\Sigma) = \Delta$ if and only if the following conditions hold:

1. $\forall x, y \in Element_0(sort_{01}(x) = sort_{01}(x) \in L \Rightarrow x = y)$.

2. $\forall x \in L(\exists y \in Element_0(sort_{01}(y) = x))$.

3. $\forall x, y \in Element_0((sort_{01}(x) \in L \;\&\; sort_{01}(y) \in L \;\&\; sort_{01}(x) \leq sort_{01}(y)) \Rightarrow x \leq y)$.

Proof It is straightforward to check that for any model Σ, the system state $C_{10}(\Sigma)$ satisfies all of the conditions. Conversely, suppose that Δ satisfies all of the conditions. Define sets as follows:

$$E = sort_{01}^{-1}(Element_2) \quad L = sort_{01}^{-1}(Literal) \quad P = sort_{01}^{-1}(Property_2)$$

Conditions 1 and 2 imply that $sort_{01}$ maps L isomorphically onto *Literal* as sets. Condition 3 implies that this is an isomorphism of partially ordered sets. Accordingly, one may identify the partially ordered sets L and *Literal*.

Putting all of the above together, the 5-tuple $(E, L, P, domain_0, range_0)$ satisfies all of the conditions for a model. It remains to show that $C_{10}(E, L, P, domain_0, range_0) = \Delta$.

1. $Element_0^C = E + L$, and $Element_0 = E \cup L$. By construction, the sets E and L are disjoint so $Element_0^C = Element_0$.

2. $Property_0^C = Property_0$ by definition.

3. $sort_{01}^C$ and $sort_{01}$ coincide by the construction of the sets E, L, P.

4. The functions $domain_0^C$ and $domain_0$ coincide by construction, and the same is true of the functions $range_0^C$ and $range_0$.

■

7 Formalization of the MOF

The layers of the CMF have a natural relationship with each other, since each model is based on a specific metamodel and each system state is based on a specific model. Putting these together forms the following diagram:

$$\begin{array}{c} \mathbf{M}_3^C \\ \uparrow \mathcal{U}_{23} \\ \mathbf{M}_2^C \\ \uparrow \mathcal{U}_{12} \\ \mathbf{M}_1^C \\ \uparrow \mathcal{U}_{01} \\ \mathbf{M}_0^C \end{array}$$

At the M3 layer there is exactly one entity, so the mapping $\mathcal{U}_{23}$ maps every metamodel to this singular entity. The mapping $\mathcal{U}_{12}$ maps a model to the metamodel on which it is based, and the mapping $\mathcal{U}_{01}$ maps a system state to the model on which it is based.

The reification embeddings C_{ij} go in the other direction, mapping a metamodel to a model and a model to a system state. Furthermore, the metamodel *CMF* can be regarded as another example of such a mapping as it describes the singular entity of the M3 layer. This mapping will be written C_{32}. Combining all of these mappings yields the following diagram in which the two columns are the same as the diagram above:

$$\begin{array}{ccc}
\mathbf{M}_3^C & & \mathbf{M}_3^C \\
\uparrow \mathcal{U}_{23} & \searrow C_{32} & \mathcal{U}_{23} \uparrow \\
\mathbf{M}_2^C & & \mathbf{M}_2^C \\
\uparrow \mathcal{U}_{12} & \searrow C_{21} & \mathcal{U}_{12} \uparrow \\
\mathbf{M}_1^C & & \mathbf{M}_1^C \\
\uparrow \mathcal{U}_{01} & \searrow C_{10} & \mathcal{U}_{01} \uparrow \\
\mathbf{M}_0^C & & \mathbf{M}_0^C
\end{array}$$

The diagram above commutes. We call it the *self-description embedding* of CMF. Example 4.2 and Theorems 5.4, 6.4 characterize precisely how the CMF is self-described. The self-description embedding has many more properties that will be introduced and proven in a sequel to this paper.

The next step is to characterize the MOF in the CMF, in much the same way. In other words, we construct the following commutative diagram and specify axioms that characterize those entities in the CMF that describe entities of the MOF:

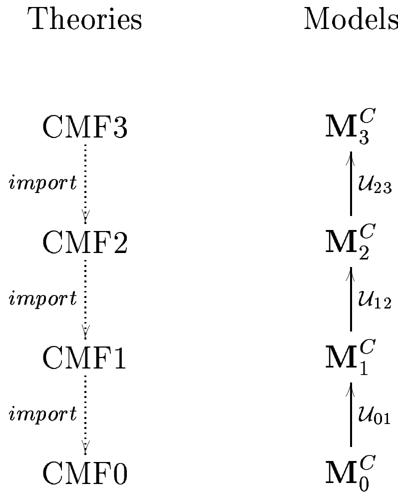
$$\begin{array}{ccc}
\mathbf{M}_3^M & & \mathbf{M}_3^C \\
\uparrow \mathcal{U}_{23} & \searrow M_{32} & \mathcal{U}_{23} \uparrow \\
\mathbf{M}_2^M & & \mathbf{M}_2^C \\
\uparrow \mathcal{U}_{12} & \searrow M_{21} & \mathcal{U}_{12} \uparrow \\
\mathbf{M}_1^M & & \mathbf{M}_1^C \\
\uparrow \mathcal{U}_{01} & \searrow M_{01} & \mathcal{U}_{01} \uparrow \\
\mathbf{M}_0^M & & \mathbf{M}_0^C
\end{array}$$

Unlike the CMF, there is no mathematical formulation of the MOF such as the left column in the diagram above, so the functions M_{ij} can only be specified informally, but the conditions characterizing the MOF are expressed formally. The entities that satisfy these conditions are the formalized MOF entities, and the collections defined by these conditions formalize the MOF. We call these conditions the MOF *axioms.* These axioms are given in Appendix A.

8 Axiomatic Formulation

The CMF is defined in sections 4, 5 and 6, using a model-theoretic formulation. We now discuss axiomatic specifications of the CMF. Most specification languages are sufficient for this purpose. We use the Slang specification language of the Specware formal methods system mainly because it has superior theory management features compared with the other major specification languages. The detailed specifications are given in Appendix B. These specifications formally restate definitions 4.1, 5.1 and 6.1 in Slang.

Each of sections 4, 5 and 6 is specified using a theory specification called CMF2, CMF1 and CMF0, respectively. In addition, the theory specification for the M3 layer is called CMF3. Each of these theory specifications imports the previous one. A theory importation defines a *theory morphism*. On the model side, each such importation results in a model of the previous kind appearing as the first component of the model. This is a general consequence of the importation of one theory in another. The following diagram summarizes this:



In this diagram, the dotted arrows are theory morphisms, and each set of models on the right side is the set of models of the corresponding theory on the left side. The import theory morphisms determine the functions $\mathcal{U}_{ij}$ in a contravariant manner.

So far, the four specifications CMF0 to CMF3 are simply four theories whose models have been explicitly described. For these four specifications to define a metamodeling facility, we need to define two additional structures:

1. The models on one layer must be the theories for the models on the next lower layer (as in Figure 1).

2. The metamodeling facility must be self-describing.

We proved the self-description property by introducing functions C_{ii-1} and proving theorems 5.4 and 6.4. Therefore the second requirement has been satisfied.

To satisfy the first requirement above there must be a mechanism for interpreting a model as a theory such that the models of this theory are exactly the ones we expect. In other

words, there must be a function that maps models to theories. This function is written $Th_i\colon \mathbf{M}_i^C \to \mathbf{Spec}$, and it is defined in Appendix C. In terms of the functions Th_i, the first requirement above is that the set of models of a theory $Th_i(m)$ be the set of models on layer $i-1$ that are based on m. In other words,

Theorem 8.1 *Let $i \in \{1,2,3\}$ and let $m \in \mathbf{M}_i^C$ be a model. Then $Mod(Th_i(m)) = \{s \in \mathbf{M}_{i-1}^C \mid \mathcal{U}_{i-1,i}(s) = m\}$.*

Proof

9 Conclusion

10 Future Work

As mentioned in the Abstract, we plan to add the necessary detail to our formal framework such that consistency checking implementations of model transformation can be demonstrated. This detail consists of completing the axioms and including the OCL that describe dynamic/evolving constraints outlined in the appendix and adding operations for these axioms. We will show transformations in terms of morphisms between formal presentations of models. We will also implement this framework in Specware, a Kestrel tool supporting category theory and algebraic logic and integrating a variety of theorem provers, to demonstrate model transformation consistency checking. We also plan to add semantic-level checking to the syntactic-level checking described in this paper. At every stage of progress, we will share our work with the OMG through presentations and the UML Infrastructure and MOF list servers and integrate comments in subsequent papers. Our plan is to make this contribution to MOF 2.0 (RFP and eventual proposal) and to contribute to any of the teams who are making UML Infrastructure submittals.

We are also enriching our framework with constructs from category theory and the theory of institutions. The layers are given a natural category structure and the embeddings C_{ij} (and M_{ij}) will be shown to be functors between these categories. The fact that these embeddings are functors means that they are not only descriptions but also *representions*. Furthermore, we show that OCL constraints can be transfered via the embeddings C_{ij} (and M_{ij}) so that constraints (as well as queries and behavior) can be computed in a different framework. The transferring of statements from one framework to another is a feature of the theory of institutions, and we will show that the layers of a metamodeling framework are institutions.

Acknowledgements

References

[1] G. Booch, I. Jacobsen, and J. Rumbaugh. *OMG Unified Modeling Language Specification*, March 2000. Available at `www.omg.org/technology/documents/formal/-`

unified_modeling_language.htm.

[2] D. Chang. *Common Warehouse Metamodel*, January 2000. Available at www.omg.org/-technology/cwm/index.htm.

[3] Object Management Group. *Common Object Request Broker Architecture (CORBA) Specifications*, December 2000. Available at www.omg.org/technology/documents/-formal/index.htm.

[4] Object Management Group. *Meta Object Facility (MOF) Specification, Version 1.3*, April 2000. Available at www.omg.org/technology/documents/formal/meta.htm.

[5] M. Richters and M. Gogolla. On formalizing the UML Object Constraint Language OCL, 1998.

A MOF Axioms

Many of the MOF Axioms follow a similar pattern, differing only in the specific elements of $Property_2^M$ or $Element_2^M$ that are being constrained. Accordingly, we begin by listing the patterns we will use.

1. **Relational**(P) is
   $\forall p, q \in Property_1^M$
   $sort_{12}^M(p) = sort_{12}^M(q) = P$ **&**
   $domain_1^M(p) = domain_1^M(q)$ **&**
   $range_1^M(p) = range_1^M(q)$
   $\Rightarrow p = q.$

2. **MandatoryDomain**(P) is
   $\forall x \in Element_1^M$
   $sort_{12}^M(x) \leq domain_2^M(P) \Rightarrow$
   $\exists p \in Property_1^M$
   $sort_{12}^M(p) \leq P$ **&** $domain_1^M(p) = x.$

3. **MandatoryRange**(P) is
   $\forall x \in Element_1^M$
   $sort_{12}^M(x) \leq range_2^M(P) \Rightarrow$
   $\exists p \in Property_1^M$
   $sort_{12}^M(p) \leq P$ **&**
   $range_1^M(p) = x.$

4. **UniqueDomain**(P) is
   $\forall p, q \in Property_1^M$
   $sort_{12}^M(p) \leq P$ **&** $sort_{12}^M(q) \leq P$ **&** $domain_1^M(p) = domain_1^M(q)$
   $\Rightarrow p = q.$

5. **UniqueRange**(P) is
   $\forall p, q \in Property_1^M$
   $sort_{12}^M(p) \leq P$ **&** $sort_{12}^M(q) \leq P$ **&** $range_1^M(p) = range_1^M(q)$
   $\Rightarrow p = q.$

6. **Functional**(P) is
   MandatoryDomain(P) **&** **UniqueDomain**(P).

7. **Contains**(P, Q) is
   $\forall p, q, r \in Property_1^M$
   $sort_{12}^M(p) \leq P$ **&** $sort_{12}^M(q) \leq Q$ **&** $sort_{12}^M(r) \leq P$ **&**
   $domain_1^M(p) = domain_1^M(q)$ **&** $range_1^M(q) = domain_1^M(r)$
   $\Rightarrow range_1^M(p) = range_1^M(r).$

8. **TransitiveClosure**(P, x, y) is
   $\exists p \in Property_1^M$
   $sort_{12}^M(p) \leq P$ **&** $domain_1^M(p) = x$ **&** $range_1^M(p) = y$ **or**
   $\exists z \in Element_1^M$
   TransitiveClosure(P, x, z) **&** **TransitiveClosure**(P, z, y).

9. **Covers**(P, Q) is
   $\forall p, q \in Property_1^M$
   $sort_{12}^M(p) \leq Q$ **&** $sort_{12}^M(q) \leq Q$ **&**
   $range_1^M(p) = range_1^M(q)$ **&** $p \neq q$
   $\Rightarrow$ **TransitiveClosure**$(P, domain_1^M(p), domain_1^M(q))$.

The following are the MOF axioms:

1. The annotation meta-attribute of ModelElement is mandatory and unique (3.4.1).
   Functional(*annotation*).

2. The name meta-attribute of ModelElement is mandatory and unique (3.4.1).
   Functional(*name*).

3. The isAbstract meta-attribute of GeneralizableElement is mandatory and unique (3.4.3).
   Functional(*isAbstract*).

4. The isLeaf meta-attribute of GeneralizableElement is mandatory and unique (3.4.3).
   Functional(*isLeaf*).

5. The isRoot meta-attribute of GeneralizableElement is mandatory and unique (3.4.3).
   Functional(*isRoot*).

6. The visibility meta-attribute of GeneralizableElement is mandatory and unique (3.4.3).
   Functional(*visibility*).

7. The isSingleton meta-attribute of Class is mandatory and unique (3.4.6).
   Functional(*isSingleton*).

8. The typeCode meta-attribute of DataType is mandatory and unique (3.4.7).
   Functional(*typeCode*).

9. The scope meta-attribute of Feature is mandatory and unique (3.4.9).
   Functional(*scope*).

10. The visibility meta-attribute of Feature is mandatory and unique (3.4.9).
    Functional(*visibility*).

11. The isChangeable meta-attribute of StructuralFeature is mandatory and unique (3.4.10).
    Functional(*isChangeable*).

12. The multiplicity meta-attribute of StructuralFeature is mandatory and unique (3.4.10). **Functional**(*multiplicity*).

13. The isDerived meta-attribute of Attribute is mandatory and unique (3.4.11). **Functional**(*isDerived*).

14. The isQuery meta-attribute of Operation is mandatory and unique (3.4.14). **Functional**(*isQuery*).

15. The isDerived meta-attribute of Association is mandatory and unique (3.4.16). **Functional**(*isDerived*).

16. The aggregation meta-attribute of AssociationEnd is mandatory and unique (3.4.17). **Functional**(*aggregation*).

17. The isChangeable meta-attribute of AssociationEnd is mandatory and unique (3.4.17). **Functional**(*isChangeable*).

18. The isNavigable meta-attribute of AssociationEnd is mandatory and unique (3.4.17). **Functional**(*isNavigable*).

19. The multiplicity meta-attribute of AssociationEnd is mandatory and unique (3.4.17). **Functional**(*multiplicity*).

20. The isClustered meta-attribute of Import is mandatory and unique (3.4.19). **Functional**(*isClustered*).

21. The visibility meta-attribute of Import is mandatory and unique (3.4.19). **Functional**(*visibility*).

22. The direction meta-attribute of Parameter is mandatory and unique (3.4.20). **Functional**(*direction*).

23. The multiplicity meta-attribute of Parameter is mandatory and unique (3.4.20). **Functional**(*multiplicity*).

24. The evaluationPolicy meta-attribute of Constraint is mandatory and unique (3.4.21). **Functional**(*evaluationPolicy*).

25. The expression meta-attribute of Constraint is mandatory and unique (3.4.21). **Functional**(*expression*).

26. The language meta-attribute of Constraint is mandatory and unique (3.4.21). **Functional**(*language*).

27. The value meta-attribute of Constant is mandatory and unique (3.4.21). **Functional**(*value*).

28. The tagID meta-attribute of Tag is mandatory and unique (3.4.23).
    Functional(*tagID*).

29. The *Contains* meta-association was reified (3.5.1).
    Functional(*container*) **&**
    Functional(*containedElement*) **&**
    UniqueDomain(*nextContainedElement*).

30. The *Contains* meta-association is an aggregation (3.5.1).
    UniqueRange(*containedElement*).

31. The Contains meta-association is ordered (3.5.1).
    UniqueRange(*nextContainedElement*) **&**
    Covers(*nextContainedElement*, *container*) **&**
    Contains(*container*, *nextContainedElement*).

32. The *Generalizes* meta-association was reified (3.5.2).
    Functional(*subtype*) **&**
    Functional(*supertype*) **&**
    UniqueDomain(*nextSupertype*).

33. The Generalizes meta-association is ordered (3.5.2).
    UniqueRange(*nextSupertype*) **&**
    Covers(*nextSupertype*, *subtype*) **&**
    Contains(*subtype*, *nextSupertype*).

34. The RefersTo meta-association is unique and mandatory for Reference. (3.5.3).
    UniqueDomain(*referent*) **&** **MandatoryDomain**(*referent*).

35. The IsOfType meta-association is unique and mandatory for TypedElement (3.5.5).
    Functional(*typedElement*).

36. The *CanRaise* meta-association was reified (3.5.6).
    Functional(*operation*) **&** **Functional**(*except*) **&**
    UniqueDomain(*nextExcept*).

37. The CanRaise meta-association is ordered (3.5.6).
    UniqueRange(*nextExcept*) **&** **Covers**(*nextExcept*, *operation*) **&**
    Contains(*operation*, *nextExcept*).

38. The Aliases meta-association is unique and mandatory for Import (3.5.7).
    Functional(*aliases*).

39. The Constrains meta-association is relational (3.5.8).
    Relational(*constrains*).

40. The Constrains meta-association is mandatory for a Constraint (3.5.8).
MandatoryDomain(*constrains*).

41. The AttachesTo meta-association is mandatory for a Tag (3.5.10).
MandatoryRange(*tag*).

42. The *AttachesTo* meta-association was reified (3.5.10).
Functional(*model*) &
Functional(*tag*) &
UniqueDomain(*nextTag*).

43. The AttachesTo meta-association is ordered (3.5.10).
UniqueRange(*nextTag*) &
Covers(*nextTag*, *model*) &
Contains(*model*, *nextTag*).

44. A ModelElement that is not a Package must be in a container (3.9.4.C-1).
$\forall x \in Element_1^M$
$sort_{12}^M(x) \leq ModelElement$ & $sort_{12}^M(x) \not\leq Package \Rightarrow$
$\exists p \in Property_1^M$
$sort_{12}^M(p) \leq containedElement$ & $range_1^M(p) = x$.

45. The names of the contents of a Namespace must not collide (3.9.4.C-5).
$\forall x, y, z \in Element_1^M, n \in Literal_v^M$
$(container^t; containedElement)(x, y)$ &
$(container^t; containedElement)(x, z)$ &
$name(y, n)$ & $name(z, n)$
$\Rightarrow y = z$.

46. A GeneralizableElement cannot be its own direct or indirect supertype (3.9.4.C-6).
$\forall x \in Element_1^M$
not TransitiveClosure$(subtype^t; subtype, x, x)$.

47. A subtype of a GeneralizableElement must be of the same kind as the GeneralizableElement itself (3.9.4.C-7).
$\forall p, q \in Property_1^M$
$domain_1^M(p) = domain_1^M(q) \Rightarrow$
$sort_{12}^M(range_1^M(p)) = sort_{12}^M(range_1^M(q))$.

48. The names of the contents of a GeneralizableElement should not collide with the names of the contents of any direct or indirect supertype (3.9.4.C-8).
$\forall x, y, z \in Element_1^M, n \in Literal_v^M$
(**TransitiveClosure**$(subtype^t; supertype); container^t; containedElement)(x, y)$ &
(**TransitiveClosure**$(subtype^t; supertype); container^t; containedElement)(x, z)$ &
$name(y, n)$ & $name(z, n)$
$\Rightarrow y = z$.

49. If a GeneralizableElement is marked as a "root," then it cannot have any supertypes (3.9.4.C-10).
$\forall x, y \in Element_1^M$
$(subtype^t; supertype)(x, y)$
$\Rightarrow isRoot(x, false)$.

50. A GeneralizableElement cannot inherit from a GeneralizableElement defined as a "leaf" (3.9.4.C-12).
$\forall x, y \in Element_1^M$
$(subtype^t; supertype)(x, y)$
$\Rightarrow isLeaf(y, false)$.

51. An Association cannot be the type of a TypedElement (3.9.4.C-13).
$\forall x, y \in Element_1^M$
$sort_{12}^M(x) \leq \mathbf{As}$ & $sort_{12}^M(y) \leq \mathbf{TE}$
$\Rightarrow$ **not** $isOfType(x, y)$.

52. A Class may contain only Classes, DataTypes, Attributes, References, Operations, Exceptions, Constraints and Tags (3.9.4.C-15).
$\forall x, y \in Element_1^M$
$sort_{12}^M(x) \leq \mathbf{C}$ & $container^t; containedElement)(x, y)$
$\Rightarrow sort_{12}^M(y) \leq \{\mathbf{C, DT, At, R, O, E, Ct, T}\}$.

53. A Class that is marked as abstract cannot also be marked as singleton (3.9.4.C-16).
$\forall x \in Element_1^M$
$sort_{12}^M(x) \leq \mathbf{C}$ & $isAbstract(x, true)$
$\Rightarrow isSingleton(x, false)$.

54. A DataType may contain only TypeAliases, Constraints and Tags (3.9.4.C-17).
$\forall x, y \in Element_1^M$
$sort_{12}^M(x) \leq \mathbf{DT}$ & $(container^t; containedElement)(x, y)$
$\Rightarrow sort_{12}^M(y) \leq \{\mathbf{TA, Ct, T}\}$.

55. The typeCode of a DataType must denote a CORBA 2.2 compliant object type or data type (3.9.4.C-18).
$\forall x \in Element_1^M, y \in Literal^M$
$sort_{12}^M(x) \leq \mathbf{DT}$ & $typeCode(x, y)$
$\Rightarrow y \notin \{$`#tk_void`, `#tk_Principal`, `#tk_null`, `#tk_except`, `#tk_value`,
`#tk_value_box`, `#tk_native`, `#tk_abstract_interface`$\}$.

56. Inheritance / generalization is not applicable to DataTypes (3.9.4.C-19).
$\forall x, y \in Element_1^M$
$(subtype^t; supertype)(x, y)$
$\Rightarrow$ **not** $sort_{12}^M(x) \leq \mathbf{DT}$.

57. A DataType cannot be abstract (3.9.4.C-20).
$\forall x \in Element_1^M$
$sort_{12}^M(x) \leq \mathbf{DT}$
$\Rightarrow isAbstract(x, false)$.

58. The multiplicity for a Reference must be the same as the multiplicity for the referenced AssociationEnd (3.9.4.C-21).
$\forall x, y \in Element_1^M, m, n \in Literal^M$
$sort_{12}^M(x) \leq \mathbf{R}$ & $refersTo(x, y)$ & $multiplicity_{AE}(x, m)$ & $multiplicity_{SF}(y, n)$
$\Rightarrow m = n$.

59. Classifier scoped References are not meaningful in the current M1 level computational model (3.9.4.C-22).
$\forall x \in Element_1^M$
$sort_{12}^M(x) \leq \mathbf{R}$
$\Rightarrow scope(x, \#\texttt{instance_level})$.

60. A Reference can be changeable only if the referenced AssociationEnd is also changeable (3.9.4.C-23).
$\forall x, y \in Element_1^M, m, n \in Literal^M$
$sort_{12}^M(x) \leq \mathbf{R}$ & $refersTo(x, y)$ & $isChangeable_{AE}(x, m)$ & $isChangeable_{SF}(y, n)$
$\Rightarrow m = n$.

61. The type attribute of a Reference and its referenced AssociationEnd must be the same (3.9.4.C-24).
$\forall x, y, z, w \in Element_1^M$
$sort_{12}^M(x) \leq \mathbf{R}$ & $refersTo(x, y)$ & $isOfType(x, z)$ & $isOfType(y, w)$
$\Rightarrow z = w$.

62. A Reference is only allowed for a navigable AssociationEnd (3.9.4.C-25).
$\forall x, y \in Element_1^M, m, n \in Literal^M$
$sort_{12}^M(x) \leq \mathbf{R}$ & $refersTo(x, y)$
$\Rightarrow isNavigable(y, true)$.

63. The containing Class for a Reference must be equal to or a subtype of the type of the Reference's exposed AssociationEnd (3.9.4.C-26).

64. An Operation may only contain Parameters, Constraints and Tags (3.9.4.C-28).

65. An Operation may have at most one Parameter whose direction is "return" (3.9.4.C-29).

66. An Exception may only contain Parameters and Tags (3.9.4.C-31).

67. An Exception's Parameters must all have the direction "out" (3.9.4.C-32).

68. An Association may only contain AssociationEnds, Constraints and Tags (3.9.4.C-33).

69. Inheritance / generalization is not applicable to Associations (3.9.4.C-34).

70. The values for "isLeaf" and "isRoot" on an Association must be true (3.9.4.C-35).

71. An Association cannot be abstract (3.9.4.C-36).

72. Associations must have visibility of "public" (3.9.4.C-38).

73. The type of an AssociationEnd must be Class (3.9.4.C-39).

74. The "isUnique" flag in an AssociationEnd's multiplicity must be true (3.9.4.C-40).

75. An Association canot have an aggregation semantic specified for both AssociationEnds (3.9.4.C-42).

76. A Package may only contain Packages, Classes, DataTypes, Associations, Exceptions, Constraints, Imports and Tags (3.9.4.C-43).

77. Packages cannot be declared as abstract (3.9.4.C-44).

78. It is only legal for a Package to import or cluster Packages or Classes (3.9.4.C-46).

79. Packages cannot import or cluster themselves (3.9.4.C-47).

80. Packages cannot import or cluster Packages or Classes that they contain (3.9.4.C-48).

81. Nested Packages cannot import or cluster other Packages or Classes (3.9.4.C-49).

82. Constraints, Tags, Imports, TypeAliases and Constraints cannot be constrained (3.9.4.C-50).

83. A Constraint can only constrain ModelElements that are defined by or inherited by its immediate container (3.9.4.C-51).

84. The type of a Constant and the type of its value must be the same (3.9.4.C-52).

85. The type of a Constant must be a CORBA data type that is legal for a CORBA 2.3 constant declaration (3.9.4.C-53).

86. The "lower" bound of a MultiplicityType cannot be negative "Unbounded" (3.9.4.C-54).

87. The "lower" bound of a MultiplicityType cannot exceed the "upper" (3.9.4.C-55).

88. The "upper" bound of a MultiplicityType cannot be less than 1 (3.9.4.C-56).

89. If a MultiplicityType specifies bounds of [0..1] or [1..1], the "is_ordered" and "is_unique" values must be false (3.9.4.C-57).

B CMF Theory Specifications

We take as given that there is a specification for the literal type structure, called Literal. In this specification, there is a sort, also called Literal, whose interpretation in a model is the set of literal sorts. The basic specifications that are used to build the CMF theories are the ones for partially ordered sets, functions that preserve a relation and the direct sum of sets with a relation.

```
spec Relation is
  sort R
  op rel: R * R -> Boolean
end-spec
```

```
spec POSet is
  import Relation rename { R -> P, rel -> leq }
  axiom reflexive is
    fa (p : P) leq (p,p)
  axiom antisymmetric is
    fa (p : P, q : P) leq (p,q) & leq (q,p) => p = q
  axiom transitive is
    fa (p : P, q : P, r : P) leq (p,q) & leq (q,r) => leq (p,r)
end-spec

spec RelFunction is
  import colimit of diagram
    node p is Relation rename { R -> P, rel -> relP }
    node q is Relation rename { R -> Q, rel -> relQ }
  end-diagram
  op f: P -> Q
  axiom relation-preserving is
    fa (p : P, q : P) relP(p,q) => relQ(f(p),f(q))
end-spec

spec RelSum is
  import colimit of diagram
    node a is Relation rename { R -> A, rel -> relA }
    node b is Relation rename { R -> B, rel -> relB }
    node c is Relation rename { R -> C, rel -> relC }
  end-diagram
  sort-axiom C = A + B
  axiom relation-sum is
    fa (c : C, d : C)
      relC (c,d) <=>
        (ex (ac : A, ad : A)
           relA (ac,ad) & ins(A)(ac) = c & ins(A)(ad) = d) or
        (ex (bc : B, bd : B)
           relB (bc,bd) & ins(B)(bc) = c & ins(B)(bd) = d)
end-spec
```

The easiest of the four CMF theories is CMF3. It consists of no sorts and no operations. It has exactly one model, consisting of no sets and no functions.

```
spec CMF3 is
end-spec
```

The next CMF theory is constructed by using a colimit to impose the axioms for partially ordered sets, order-preserving functions and the direct sum of partially ordered sets. It begins with just the two sorts Element2 and Property2:

```
spec CMF2base0 is
  sort Element2
  sort Property2
end-spec
```

In the next step, we make Element2 and Property2 into partially ordered sets, and we add CMF3 and Literals.

```
spec CMF2base1 is
  colimit of diagram
    node cmf3import is CMF3
    node literalimport is Literal rename { leq -> leqL }
    node base is CMF2Base0
    node e2poset is POSet rename { leq -> leqE2 }
    node p2poset is POSet rename { leq -> leqP2 }
    node t1 is Triv
    node t2 is Triv
    arc t1 -> e2poset is { T -> P }
    arc t1 -> base is { T -> Element2 }
    arc t2 -> p2poset is { T -> P }
    arc t2 -> base is { T -> Property2 }
  end-diagram
end-spec
```

Next we add the direct sum of the Element2 and Literal both of which are now partially ordered sets.

```
spec CMF2base2 is
  colimit of diagram
    node sum is RelSum rename { C -> EL2, relC -> leqEL2 }
    node base is CMF2base1
    node t1 is Relation
    node t2 is Relation
    arc t1 -> sum is { R -> A, rel -> relA }
    arc t1 -> base is { R -> Element2, rel -> leqE2 }
    arc t2 -> sum is { R -> B, rel -> relB }
    arc t2 -> base is { R -> Literal, rel -> leqL }
  end-diagram
```

Finally, the domain2 and range2 operations are introduced.

```
spec CMF2 is
  colimit of diagram
    node base is CMF2base2
```

```
    node d2 is RelFunction rename { f -> domain2 }
    node r2 is RelFunction rename { f -> range2 }
    node td1 is Relation
    node td2 is Relation
    node tr1 is Relation
    node tr2 is Relation
    arc td1 -> d2 is { R -> P, rel -> relP }
    arc td1 -> base is { R -> Property2, rel -> leqP2 }
    arc td2 -> d2 is { R -> Q, rel -> relQ }
    arc td2 -> base is { R -> Element2, rel -> leqE2 }
    arc tr1 -> r2 is { R -> P, rel -> relP }
    arc tr1 -> base is { R -> Property2, rel -> leqP2 }
    arc tr2 -> r2 is { R -> Q, rel -> relQ }
    arc tr2 -> base is { R -> EL2, rel -> leqEL2 }
  end-diagram
end-spec
```

We construct CMF1 in a similar manner to CMF2, starting from a base, and using colimit for the partially ordered sets and order-preserving functions.

```
spec CMF1base0 is
  sort Element1
  sort Property1
end-spec

spec CMF1base1 is
  colimit of diagram
    node cmf2import is CMF2
    node base is CMF1base0
    node e1poset is POSet rename { leq -> leqE1 }
    node p1poset is POSet rename { leq -> leqP1 }
    node t1 is Triv
    node t2 is Triv
    arc t1 -> e1poset is { T -> P }
    arc t1 -> base is { T -> Element1 }
    arc t2 -> p1poset is { T -> P }
    arc t2 -> base is { T -> Property1 }
  end-diagram
end-spec

spec CMF1base2 is
  colimit of diagram
    node sum is RelSum rename { C -> EL1, relC -> leqEL1 }
```

```
    node base is CMF1base1
    node t1 is Relation
    node t2 is Relation
    arc t1 -> sum is { R -> A, rel -> relA }
    arc t1 -> base is { R -> Element1, rel -> leqE1 }
    arc t2 -> sum is { R -> B, rel -> relB }
    arc t2 -> base is { R -> Literal, rel -> leqL }
  end-diagram

spec CMF1 is
  import colimit of diagram
    node base is CMF1base2
    node se is RelFunction rename { f -> elementsort12 }
    node sp is RelFunction rename { f -> propertysort12 }
    node d1 is RelFunction rename { f -> domain1 }
    node r1 is RelFunction rename { f -> range1 }
    node se1 is Relation
    node se2 is Relation
    node sp1 is Relation
    node sp2 is Relation
    node td1 is Relation
    node td2 is Relation
    node tr1 is Relation
    node tr2 is Relation
    arc se1 -> se is { R -> P, rel -> relP }
    arc se1 -> base is { R -> Element1, rel -> leqE1 }
    arc se2 -> se is { R -> Q, rel -> relQ }
    arc se2 -> base is { R -> Element2, rel -> leqE2 }
    arc sp1 -> sp is { R -> P, rel -> relP }
    arc sp1 -> base is { R -> Property1, rel -> leqP1 }
    arc sp2 -> sp is { R -> Q, rel -> relQ }
    arc sp2 -> base is { R -> Property2, rel -> leqP2 }
    arc td1 -> d1 is { R -> P, rel -> relP }
    arc td1 -> base is { R -> Property1, rel -> leqP1 }
    arc td2 -> d1 is { R -> Q, rel -> relQ }
    arc td2 -> base is { R -> Element1, rel -> leqE1 }
    arc tr1 -> r1 is { R -> P, rel -> relP }
    arc tr1 -> base is { R -> Property1, rel -> leqP1 }
    arc tr2 -> r1 is { R -> Q, rel -> relQ }
    arc tr2 -> base is { R -> EL1, rel -> leqEL1 }
  end-diagram
  op el12: EL1 -> EL2
  axiom definition-of-el12 is
```

```
    (fa (e : Element1)
       el12(ins(Element1)(e)) = ins(Element2)(elementsort12(e))) &
    (fa (l : Literal)
       el12(ins(Literal)(l)) = ins(Literal)(l))
  axiom sort-compatibility-1 is
    fa (p : Property1)
      leqE2 (elementsort12(domain1(p)),domain2(propertysort12(p)))
  axiom sort-compatibility-2 is
    fa (p : Property1)
      leqEL2 (el12(range1(p)),range2(propertysort12(p)))
end-spec
```

We construct CMF0 exactly the same way as CMF1. In fact, one could continue to define layers below CMF0 because the process repeats.

```
spec CMF0base0 is
  sort Element0
  sort Property0
end-spec

spec CMF0base1 is
  colimit of diagram
    node cmf1import is CMF1
    node base is CMF0base0
    node e0poset is POSet rename { leq -> leqE0 }
    node p0poset is POSet rename { leq -> leqP0 }
    node t1 is Triv
    node t2 is Triv
    arc t1 -> e0poset is { T -> P }
    arc t1 -> base is { T -> Element0 }
    arc t2 -> p0poset is { T -> P }
    arc t2 -> base is { T -> Property0 }
  end-diagram
end-spec

spec CMF0base2 is
  colimit of diagram
    node sum is RelSum rename { C -> EL0, relC -> leqEL0 }
    node base is CMF0base1
    node t1 is Relation
    node t2 is Relation
    arc t1 -> sum is { R -> A, rel -> relA }
    arc t1 -> base is { R -> Element0, rel -> leqE0 }
```

```
    arc t2 -> sum is { R -> B, rel -> relB }
    arc t2 -> base is { R -> Literal, rel -> leqL }
  end-diagram

spec CMF0 is
  import colimit of diagram
    node base is CMF0base2
    node se is RelFunction rename { f -> elementsort01 }
    node sp is RelFunction rename { f -> propertysort01 }
    node d0 is RelFunction rename { f -> domain0 }
    node r0 is RelFunction rename { f -> range0 }
    node se1 is Relation
    node se2 is Relation
    node sp1 is Relation
    node sp2 is Relation
    node td1 is Relation
    node td2 is Relation
    node tr1 is Relation
    node tr2 is Relation
    arc se1 -> se is { R -> P, rel -> relP }
    arc se1 -> base is { R -> Element0, rel -> leqE0 }
    arc se2 -> se is { R -> Q, rel -> relQ }
    arc se2 -> base is { R -> Element1, rel -> leqE1 }
    arc sp1 -> sp is { R -> P, rel -> relP }
    arc sp1 -> base is { R -> Property0, rel -> leqP0 }
    arc sp2 -> sp is { R -> Q, rel -> relQ }
    arc sp2 -> base is { R -> Property1, rel -> leqP1 }
    arc td1 -> d0 is { R -> P, rel -> relP }
    arc td1 -> base is { R -> Property0, rel -> leqP0 }
    arc td2 -> d0 is { R -> Q, rel -> relQ }
    arc td2 -> base is { R -> Element0, rel -> leqE0 }
    arc tr1 -> r0 is { R -> P, rel -> relP }
    arc tr1 -> base is { R -> Property0, rel -> leqP0 }
    arc tr2 -> r0 is { R -> Q, rel -> relQ }
    arc tr2 -> base is { R -> EL0, rel -> leqEL0 }
  end-diagram
  op el01: EL0 -> EL1
  axiom definition-of-el01 is
    (fa (e : Element0)
       el01(ins(Element0)(e)) = ins(Element1)(elementsort01(e))) &
    (fa (l : Literal)
       el01(ins(Literal)(l)) = ins(Literal)(l))
  axiom sort-compatibility-1 is
```

```
    fa (p : Property0)
      leqE1 (elementsort01(domain0(p)),domain1(propertysort01(p)))
  axiom sort-compatibility-2 is
    fa (p : Property0)
      leqEL1 (el01(range0(p)),range1(propertysort01(p)))
end-spec
```

C Theory Mapping

The theory mappings defined in this appendix construct a theory from a model. This is an essential part of a metamodeling facility, since the ordinary theory/model paradigm has just two layers, while a metamodeling facility has more than two. In this section we define the theory mappings Th_3, Th_2 and Th_1. One could also define Th_0, but this would only be useful if there were layers lower than layer 0.

The simplest theory mapping is Th_3 since there is exactly one model on layer 3. We define Th_3 on this singular model to be the theory whose specification is CMF2 in Appendix B.

Next consider Th_2. Let $\Omega = (\mathit{Element}_2, \mathit{Literal}, \mathit{Property}_2, \mathit{domain}_2, \mathit{range}_2)$ be a metamodel. We define $Th_2(\Omega)$ by starting with the theory

```
spec Th2Omega is
  import CMF2
end-spec
```

and adding sorts, operations and axioms. In the following, we presume that one can choose appropriate unique names for the various sorts and operations. Thus when we write an expression such as `E2(e)`, the intention is that this represents the unique name chosen for the use of e in this context.

1. For every $e \in \mathit{Element}_2$, add a sort `E2(e)`.
2. For every $p \in \mathit{Property}_2$, add a sort `P2(e)`.
3. For every pair $e, f \in \mathit{Element}_2$ such that $e < f$,
   add an operation `incl(e,f): E2(e) -> E2(f)` and the
   `axiom fa(x:E2(e),y:E2(e)) incl(e,f)(x) = incl(e,f)(y) => x = y`.
4. For every triple $e, f, g \in \mathit{Element}_2$ such that $e < f < g$, add the
   `axiom fa(x:E2(e)) incl(f,g)(incl(e,f)(x)) = incl(e,g)(x)`.
5. For every pair $p, q \in \mathit{Property}_2$ such that $p < q$,
   add an operation `incl(p,q): P2(p) -> P2(q)` and the
   `axiom fa(x:P2(p),y:P2(p)) incl(p,q)(x) = incl(p,q)(y) => x = y`.
6. For every triple $p, q, r \in \mathit{Property}_2$ such that $p < q < r$, add the
   `axiom fa(x:P2(p)) incl(q,r)(incl(p,q)(x)) = incl(p,r)(x)`.

7. For every $p \in Property_2$, let $d = domain_2(p)$, and add an operation `d2(p): P2(p) -> E2(d)`.

8. For every $p \in Property_2$, let $r = range_2(p)$. If $r \in Element_2$, then add an operation `r2(p): P2(p) -> E2(r)`. If $r \in Literal$, then add an operation `r2(p): P2(p) -> r`.

Finally consider Th_1. Let $\Sigma = (\Omega, Element_1, Property_1, sort_{12}, domain_1, range_1)$ be a model based on Ω. We define $Th_1(\Sigma)$ by starting with the theory

```
spec Th1Omega is
  import Th2Omega
end-spec
```

and adding sorts, operations and axioms as follows:

1. For every $e \in Element_1$, add a sort `E1(e)`.

2. For every $p \in Property_1$, add a sort `P1(e)`.

3. For every pair $e, f \in Element_1$ such that $e < f$, add an operation `incl(e,f): E1(e) -> E1(f)` and the `axiom fa(x:E1(e),y:E1(e)) incl(e,f)(x) = incl(e,f)(y) => x = y`.

4. For every triple $e, f, g \in Element_1$ such that $e < f < g$, add the `axiom fa(x:E1(e)) incl(f,g)(incl(e,f)(x)) = incl(e,g)(x)`.

5. For every pair $p, q \in Property_1$ such that $p < q$, add an operation `incl(p,q): P1(p) -> P1(q)` and the `axiom fa(x:P1(p),y:P1(p)) incl(p,q)(x) = incl(p,q)(y) => x = y`.

6. For every triple $p, q, r \in Property_1$ such that $p < q < r$, add the `axiom fa(x:P1(p)) incl(q,r)(incl(p,q)(x)) = incl(p,r)(x)`.

7. For every $p \in Property_1$, let $d = domain_1(p)$, and add an operation `d1(p): P1(p) -> E1(d)`.

8. For every $p \in Property_1$, let $r = range_1(p)$. If $r \in Element_1$, then add an operation `r1(p): P1(p) -> E1(r)`. If $r \in Literal$, then add an operation `r1(p): P1(p) -> r`.