Softw Syst Model (2008) 7:103–124
DOI 10.1007/s10270-007-0049-9

REGULAR PAPER

A metamodeling language supporting subset and union properties

Marcus Alanen · Ivan Porres

Received: 4 September 2006 / Revised: 20 December 2006 / Accepted: 20 February 2007 / Published online: 14 June 2007

Abstract In this article, we describe successive versions of a metamodeling language using a set-theoretic formalization. We focus on language extension mechanisms, particularly on the relatively new subset and union properties of MOF 2.0 and the UML 2.0 Infrastructure. We use Liskov substitutability as the rationale for our formalization. We also show that property redefinitions are not a safe language extension mechanism. Each language version provides new features, and we note how such features cannot be mixed arbitrarily. Instead, constraints over the metamodel and model structures must be established. We expect that this article provides a better understanding of the foundations of MOF 2.0, which is necessary to define new extensions, model transformation languages and tools.

Keywords Subsets · Unions · Redefinitions · UML · Package merges · Extension mechanisms · Graphs · Graph theory · Software modeling languages · Metamodeling · MOF

1 Introduction

Modeling is a fundamental approach to problem solving in all engineering disciplines, including software engineering. The role of software modeling and software modeling languages has become more significant in the software industry thanks to initiatives such as the Unified Modeling Language [39] and the Model Driven Architecture [43]. These two approaches are defined by a series of standards created and maintained by the Object Management Group (OMG).

The OMG software and system modeling standards are based on the concept of metamodeling. A metamodel is a description of a modeling language. A metamodeling language is thus a language used to describe metamodels. The Meta Object Facility 1.4 (MOF) [35], the Eclipse Modeling Framework (EMF) [16,20] and the Graph eXchange Language (GXL) [55] are well-known examples of metamodeling languages. Most of the concepts found in these languages are strikingly similar since they are all based to some extent on the object-oriented (OO) software paradigm. In these approaches, a metamodel is defined as a collection of classes and properties while a model is an instance of such classes and properties.

However, two new metamodeling languages have recently emerged with the advent of the UML 2.0 Superstructure [39]: MOF 2.0 [41] and the UML 2.0 Infrastructure [42]. These two metamodeling languages share most of the features of previous languages such as MOF 1.4, but they also introduce several new concepts not found in traditional modeling and OO programming languages, mainly: *subset* properties, *strict union* properties and property *redefinitions*. These new concepts can be used to define a new modeling language as an extension of an existing one. This is exploited in the definition of UML 2.0 itself, where the language is defined as a relatively small core that is extended and specialized into different modeling views or diagrams.

Unfortunately, very little is told in the standards [41,42] about the actual meaning of these new features. This is a critical omission since these concepts are heavily used in the

Communicated by Dr. Jean-Michel Bruel.

M. Alanen (✉) · I. Porres
TUCS Turku Centre for Computer Science,
Department of Information Technologies,
Åbo Akademi University,
Joukahaisenkatu 3–5, 20520 Turku, Finland
e-mail: marcus.alanen@abo.fi

I. Porres
e-mail: ivan.porres@abo.fi

definition of UML 2.0. A precise definition of the UML 2.0 metamodel is necessary in order to ensure interoperability of software modeling tools, such as model editors, model transformation and code generation tools.

In this article, we present a set-theoretic formalization of a metamodeling language that supports what we consider to be the core structural features of MOF 2.0 and the UML 2.0 Infrastructure, including multiple class specialization and the new subset properties. The work presented in this article can also be applied to the definition of new domain-specific modeling languages (DSML) [21]. Although this article presents a theoretical framework, we believe it represents an important contribution that can influence the practical implementation of model repositories and transformation tools for UML 2.0 and other languages.

This paper is an extended and thoroughly revised version of the paper presented in [4]. The research presented in this article is based on the study of the relevant OMG documents and research papers on related topics and also on the experiences obtained by developing an experimental modeling tool. A comparison of the main modeling languages studied by us can be found in [1]. Also, the experimental modeling tool Coral [2] has been extended to implement and validate the ideas presented in this article and it is available as open source.

We proceed as follows: Sect. 2 describes informally the new language extension mechanisms and presents the main motivations for our work. Section 3 presents the most basic formalism that will be developed and extended during the paper. Section 4 introduces property characteristics such as multiplicity and composition, while Sects. 5 and 6 deal with class and property specialization, edf, respectively. Alternative approaches to property specialization proposed by other authors such as covariant specialization and property redefinition are described in Sect. 7. Finally, Sect. 8 contains related work while Sect. 9 contains some concluding remarks. We also provide two appendices: Appendix A summarizes the final metamodeling language, and Appendix B summarizes the mathematical notation used in the paper.

2 Extension mechanisms in MOF 2.0 and the UML 2.0 Infrastructure

MOF 2.0 and the UML 2.0 Infrastructure propose mainly four extension mechanisms: class specializations, property subsets and unions, property redefinitions and package merges. Class specialization is identical to class inheritance in OO languages. A specialized class inherits all the properties of its base classes, and it can define new properties. Subset and union properties are two mechanisms to specialize a property defined in a base class. Property redefinition allows us to arbitrarily replace a property with another one. Finally, various package merges allow us to combine different documents describing different (parts of) metamodels into one. These mechanisms allow a metamodel to be developed and extended by different parties. The extension mechanisms can also be useful to define very large metamodels, such as the UML 2.0 Superstructure, even when the definition is provided by one party.

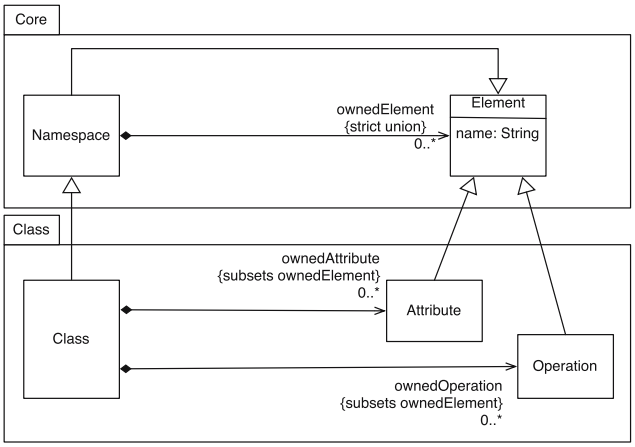

Fig. 1 Metamodel for a UML class diagram using subset and union properties

2.1 Class and property specialization

Figure 1 contains an example class diagram metamodel of the use of these concepts. The *Element* class represents any kind of element in a UML model. It has one property *name* of type string. A *Namespace* is a specialization of *Element* that can contain other elements. Since a *Namespace* is also an *Element* it also has a property called *name*. It also has a property named *ownedElement* to refer to the elements contained by it.

A *Namespace* is an abstract container that does not appear as such in any model. However, it is useful to define many other elements such as a class that can contain attributes and operations or a package that can contain classes and other packages. We can define concepts such as *Class* and *Package* as direct specializations of *Namespace*.

Each specialization of *Namespace* can contain other elements. However, the types of elements that can be contained depends on the specialization. For example, in UML a *Class* can contain attributes and operations while a *Package* can contain classes and other packages. As a consequence, the specialized classes should not use the original *ownedElement* property, because otherwise a *Class* could contain any other element such as another *Package*.

The solution is to use the new property features to specialize the *ownedElement* property of *Namespace*. In the example, we specialize *Namespace* into a *Class* and add two

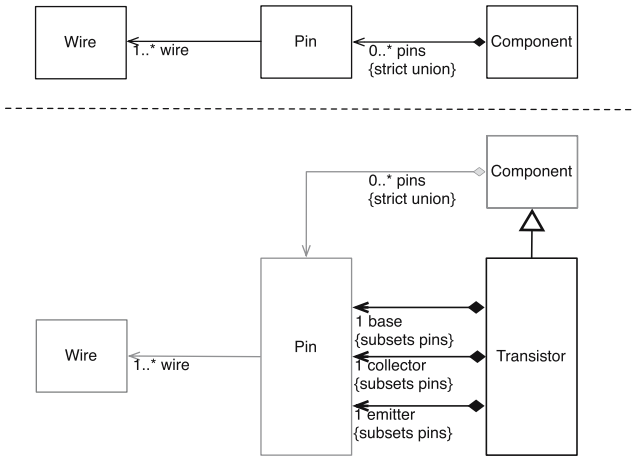

Fig. 2 *Top* Base language for electronic circuits. *Bottom* Example extension of the digital circuit metamodel by specialization of Component and its properties into Transistor

subset properties called *ownedAttribute* and *ownedOperation* to keep attributes and operations. These properties are *subsetting* the *ownedElement* property. Since the new subset properties point to *Operation* and *Attribute*, they can only contain elements of such type (or their subtypes).

Figure 2 contains an example of the use of subset properties in a domain-specific modeling language. We abstract electronic components and their interconnecting wires in a digital electronic circuit into three classes, *Wire*, *Pin* and *Component*. An example specialization of *Component* is *Transistor*, which represents a transistor connecting to three pins: *base*, *collector* and *emitter*. While a generic component may have an arbitrary number of pins, a transistor may only have three specific pins.

This example also shows that subsetting does not partition the subsetted property with respect to classes; several subset relations can be built between the same two classes. We should also note that subset properties can be useful even when they are not used in combination with class specialization. That is, we can define a property and its subsets in the same class. We should note that the *pins* property needs to be a strict union. Otherwise we could connect a transistor to other pins that are not the base, collector or emitter.

2.2 Criteria for language extensions

Most of the artifacts that compose a model-driven development method such as model transformations, model queries and code generators depend on a specific modeling language, such as the UML 2 Superstructure. Since it is now possible to extend and modify metamodels we should consider what is the impact of these extensions in model transformations, model queries and code generators.

Our main criteria for language extensions is that artifacts defined for the original language should still be usable in any of its extensions. This concept is similar to the Liskov substitutability [30] used in program type systems. This is not a surprise since a modeling language can be seen as a type for a model transformation program. As a consequence, a language extension should not be able to arbitrarily redefine or remove classes or properties from a language since existing artifacts may depend on them.

As an example of Liskov substitutability, consider Fig. 2 and a transformation that takes a *Component* as its input parameter, producing as output the various pins that the *Component* has. If we were to pass in a subclass of *Component*, for example *Transistor* as a parameter to this transformation, we would still expect it to work as intended. But if the *Transistor* class could somehow remove the *pins* property between itself and *Pin*, our transformation would fail. In that hypothetical case, a *Transistor* would not be Liskov-substitutable for a *Component*.

Specialization is a very strict and limiting concept as it implies a tight coupling between classes. Therefore, Liskov substitutability is also equally limiting. However, it does provide a clear mathematical foundation which makes reasoning about programs and models easier.

2.3 Package merge

We should note that MOF 2.0 and the UML 2.0 Infrastructure contain other related concepts to define language extensions such as package merges.

We consider that package merges, albeit important, only influence the division of a metamodel into different documents. They do not influence the relationship between model elements. A metamodel using package merges can always be transformed into a metamodel without package merges. Thereby its semantics are defined by this transformation operation and as such do not provide a new relationship construct to metamodel developers. This has already been noted by Jim Steel and Jean-Marc Jézéquel in [47]. This does not mean that package merging is not useful, just that it is not necessary to discuss it within the scope of this paper.

Therefore, we do not study in this article the concept of package merges and we focus on the semantics of subset properties since we consider that these are the main novelties in MOF 2.0 and the core mechanism for language extension.

2.4 Why do we want metamodeling languages anyway?

We have seen in the previous examples that property specialization is an interesting addition to the UML 2 Infrastructure. However, we need to consider what its impact on the UML 2 Superstructure is, or even what the role of metamodeling languages in model-based software development is. On one

hand, many UML practitioners are not even aware of the existence of the UML metamodel and therefore new additions to it such as the subset properties do not affect them directly. On the other hand, the UML Infrastructure and the UML metamodel play an important role in the development of model-based development environments, including model repository components, metamodel zoos, model transformation languages and related tools.

A model repository is a reusable software component that is used to manage models described in user-defined modeling languages. It provides the basic functionality to create models, add, retrieve, update and delete elements in an existing model and to store and retrieve models in an XML-based document. Some examples of these components are the Eclipse Modeling Framework (EMF) [20], the Netbeans Metadata Repository (NMR) [32] and Andrew Sutton's Open Modeling Framework (OMF) [49]. These components are often used by interactive editors and model transformation components in order to build complete model-driven development tools. In order to ensure interoperability between components, they should have a common understanding of the underlying metamodeling language.

A metamodel zoo such as the Atlantic Zoo [50] is an online collection of metamodels available to tool developers. The Atlantic Zoo is based on the KM3 [27] metamodeling language, but other metamodeling languages are supported using automated model transformations. A precise definition of the metamodeling languages used in a zoo is necessary in order to reuse the metamodels and to define the transformation mappings.

Also, we should note that a metamodel works as a type for model transformations [48]. Model transformation languages and tools are therefore based on the type system defined by a specific metamodeling language. Examples of model transformation languages related to the OMG modeling standards are [14,19,28,38,44,51,52].

Finally, XMI [36,37,40] is used to interchange models between modeling tools. Different authors have observed that many of the issues that appear when interchanging models in practice between tools are a consequence of different implementations of the UML metamodel by different tools [3,26,31].

3 Metamodels and models

In this section, we discuss a simple metamodeling language based on simple classes and properties. This simple language does not include any extension mechanism, but it will serve to explain the most basic concepts that appear in MOF and UML in detail. We will denote our basic language with a subscript B in the names of various structures and functions. In the following sections we will add generalizations (class specializations) and property subsets as successive features in languages G and S, respectively. We proceed in this fashion in order to simplify the exposition of these concepts in this paper. Also, we want to show how each new concept in a metamodeling language interacts with the existing ones, sometimes in rather unexpected ways.

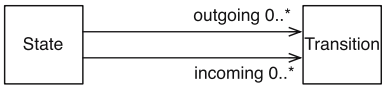

Fig. 3 A UML class diagram representing a (part of a) metamodel for Statecharts

3.1 A basic metamodeling language

Metamodels are composed of classes and properties. A class represents a kind of abstraction that can appear in a model such as a state or a transition in a Statechart [24], while a property represents a basic relationship between these abstractions such as the fact that each transition has a source state and a target state. Models, on the other hand, are described using elements and slots, where each element conforms to one single class and each slot conforms to one single property.

UML and MOF use the UML class diagram notation to describe modeling languages visually. Figure 3 shows a part of a metamodel for a Statechart that we will use as our running example. This metamodel contains two classes: *State* and *Transition*, and two properties, *outgoing* and *incoming*. These two properties belong to *State* and have *Transition* as their type. A property may also contain several annotations that we call property characteristics. In the figure, we can see the multiplicity characteristic of the properties as the label "0..*".

3.2 Metamodel formalization

Formally, we define a modeling language in our basic metamodeling framework as a tuple $ML_B = (C, P, \text{owner}, \text{type}, \text{characteristics})$ where C is a finite set of classes, P is a finite set of properties and $C \cap P = \emptyset$. In our example, the set of classes is $C = \{State, Transition\}$, while the set of properties is $P = \{incoming, outgoing\}$.

Each property has a class as an owner, and this fact is indicated by the function $\text{owner} : P \to C$. The function $\text{properties} : C \to \mathcal{P}(P)$ gives the properties that belong to a specific class such that $(\forall c \in C \cdot \text{properties}(c) = \{p \cdot c = \text{owner}(p)\})$. In our running example, we have $\text{owner}(incoming) = State$ and $\text{owner}(outgoing) = State$, while the properties of the classes are $\text{properties}(State) = \{incoming, outgoing\}$ and $\text{properties}(Transition) = \emptyset$. Finally, the function $\text{type} : P \to C$ denotes the type of

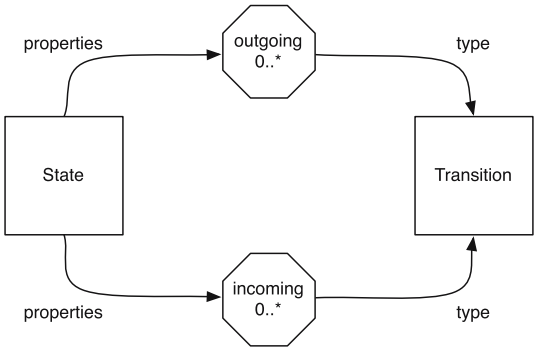

Fig. 4 The metamodel from Fig. 3 as a graph of classes and properties

elements in the property. In our example, both properties have the class *Transition* as their type, therefore type(*incoming*) = type(*outgoing*) = *Transition*. We define the characteristics of a property in detail in Sect. 4.

We can represent a modeling language ML_B as a labeled directed graph $G = (V, A, l)$. The set of vertices V is the union of the set of classes and properties, $V = C \cup P$. The set of arcs A contains two arcs for each property, one from the owner of a property to it and one from the property to its type: $A = \{(\text{owner}(p), p) \cdot p \in P\} \cup \{(p, \text{type}(p)) \cdot p \in P\}$. The property characteristics are represented as labels l over the nodes of the graph.

An example of this representation of Fig. 3 is shown in Fig. 4. To facilitate the comprehension of the graph, we represent classes as rectangles and properties as octagons. Although this notation is less compact than UML class diagrams, it maps better to the structures defined by ML_B. Additionally, we explicitly refrain from using UML in order to make it clear that UML is not a prerequisite for the metamodeling language described in this paper.

Understanding modeling languages and models as graphs brings many benefits to our approach since graph theory [45] provides a solid foundation to many modeling approaches and model transformation languages as described for example in [8,13,54].

3.3 Model formalization

We define a model as the tuple $M = (E, \text{class}, S, \text{property}, \text{slotOwner}, \text{contents})$, where E is a finite set of elements, S is a finite set of slots and $E \cap S = \emptyset$. Each slot has one element as its owner as represented by the function slotOwner : $S \rightarrow E$. For convenience, we can also define the slots of a given element as the function slots : $E \rightarrow \mathcal{P}(S)$, where $(\forall e \in E \cdot \text{slots}(e) = \{s \cdot e = \text{slotOwner}(s)\})$.

A slot may refer to a number of elements as its contents. This is represented by the function contents. The value of this function is either a set of elements if the order of elements does not matter, and thus contents : $S \rightarrow \mathcal{P}(E)$; otherwise, the function returns a sequence of elements and we say contents : $S \rightarrow (E, \prec)$. We discuss the ordered characteristic in more detail in Sect. 4.2. We define the size of a slot to be the amount of elements referenced by that slot: $(\forall s \in S \cdot \#s = \#\text{contents}(s))$.

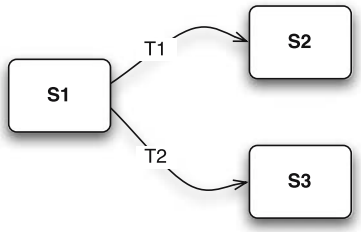

Fig. 5 An example Statechart represented in the UML notation

Figure 5 shows an example model based on a Statechart language. In the example, the set of elements is $E = \{S1, S2, S3, T1, T2\}$ and $S = \{in1, out1, in2, out2, in3, out3\}$. Also, slotOwner $= \{in1 \rightarrow S1, in2 \rightarrow S2, in3 \rightarrow S3, out1 \rightarrow S1, out2 \rightarrow S2, out3 \rightarrow S3\}$. Since the two properties are not ordered, we have contents($out1$)=$\{T1, T2\}$, contents($out2$) = contents($out3$) = contents($in1$) = $\emptyset$, contents($in2$) = $\{T1\}$ and contents($in3$) = $\{T2\}$.

Each element in a model conforms to a class in a language and each slot conforms to a property. This conformance is represented by the functions class : $E \rightarrow C$ and property : $S \rightarrow P$ in a model. Together these functions link a model to its metamodel, thereby establishing a certain set of constraints that must be satisfied for any valid model, e.g., what are the slots owned by an element, the class of the elements in a slot and the amount of elements in a slot.

If a model satisfies all these constraints, we say that the model conforms to its metamodel. We should note there is no reason to separate a model from its metamodel by disregarding the class and property functions. Such a model would merely be a graph of nodes connected by directed edges and, in most cases, would not contain enough information to be understood.

Models are usually depicted using their own concrete syntax. For example, in a UML Statechart, states are represented as rounded rectangles and transitions as arcs. In this article, we use a generic syntax where all models are represent in an uniform way, independent of their modeling language.

We can represent a model as a labeled directed graph $G = (V, A, l)$. The set of vertices V is the union of the set of elements and slots, $V = E \cup S$. The set of arcs A contains an arc for each slot and for each element reference in a slot, $A = \{(\text{slotOwner}(s), s) \cdot s \in S\} \cup \{(s, e) \cdot s \in S \wedge e \in \text{contents}(s)\}\}$. The relation between between elements and slots at the model layer are represented as labels l.

We depict each model element as rounded rectangle and each slot as a circle. Figure 6 shows the example model as such a graph.

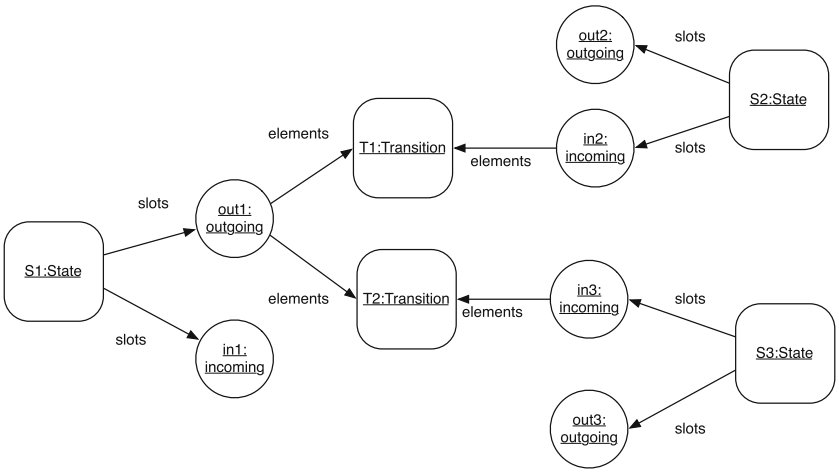

Fig. 6 An example Statechart model represented using the generic model notation

3.4 Model constraints

The effective properties of a class is the set of all properties that can be used in an element conforming to that class. In a simple modeling language that does not support class specialization, the effective properties are simply the properties defined directly by the class. In Sect. 5, we will review this definition to take into account properties defined in superclasses.

$$\text{effectiveProperties}_B(c) = \text{properties}(c), \quad \forall c \in C$$

The function $\text{effectiveProperties}_B$ is specific to the metamodeling language ML_B. Nevertheless the constraints we write use the generic function $\text{effectiveProperties} : C \to \mathcal{P}(P)$. Depending on the metamodeling language used, we can substitute different definitions in its stead. We do similarly for other functions as well, but will henceforth omit this explanation.

The effective properties of a class introduce two constraints over the elements conforming to that class. First, an element cannot have slots that do not conform to the effective properties of its class.

Model Constraint 1 *Valid slots in element* (1): $(\forall e \in E \cdot (\forall s \in \text{slots}(e) \cdot (\text{property}(s)) \in \text{effectiveProperties}(\text{class}(e))))$

Second, an element must have exactly one slot for each effective property in its class:

Model Constraint 2 *Valid slots in element* (2): $(\forall e \in E \cdot (\forall p \in \text{effectiveProperties}(\text{class}(e)) \cdot (\exists! s \in \text{slots}(e) \cdot \text{property}(s) = p)))$

The function $\text{effectiveType} : P \to \mathcal{P}(C)$ denotes the effective types of a property, i.e., the set of all allowed types for that property. In this simple language, the effective types of a property is defined explicitly by the type characteristic.

$$\text{effectiveType}_B(p) = \{\text{type}(p)\}, \quad \forall p \in P$$

The set of effective types of a property constrains the class of the elements that can be in a slot conforming to the property:

Model Constraint 3 *Class of elements in a slot*: $(\forall s \in S \cdot (\forall e \in \text{contents}(s) \cdot \text{class}(e) \in \text{effectiveType}(\text{property}(s))))$

Figure 7 shows the example metamodel for Statecharts together with a part of the example model. We have represented the class and property functions with dashed lines. Since models and metamodels are finite, we can easily check that the previous constraints hold for the example.

3.5 Metamodel constraints

Since a metamodel defines a set of constraints over a model, it can be possible to define a metamodel in such a way that there is no nontrivial model that conforms to it. Similarly, it may be possible to define a class so that there is no element that conforms to it, or a property so that there is no nontrivial slot that conforms to it. In these cases, we say that the metamodel, class or property is *void*.

Void metamodels or metamodels with void classes or properties are not useful in software development. For this reason, we will define metamodel constraints in our metamodeling approach. A metamodel constraint is a predicate over a metamodel that should hold in order to exclude void definitions.

3.6 Primitive values

Hitherto, the models that can be described can only consist of elements interconnected via slots. It is often the case that we need to use primitive data values such as strings, integers, floating-point values and enumeration values. However,

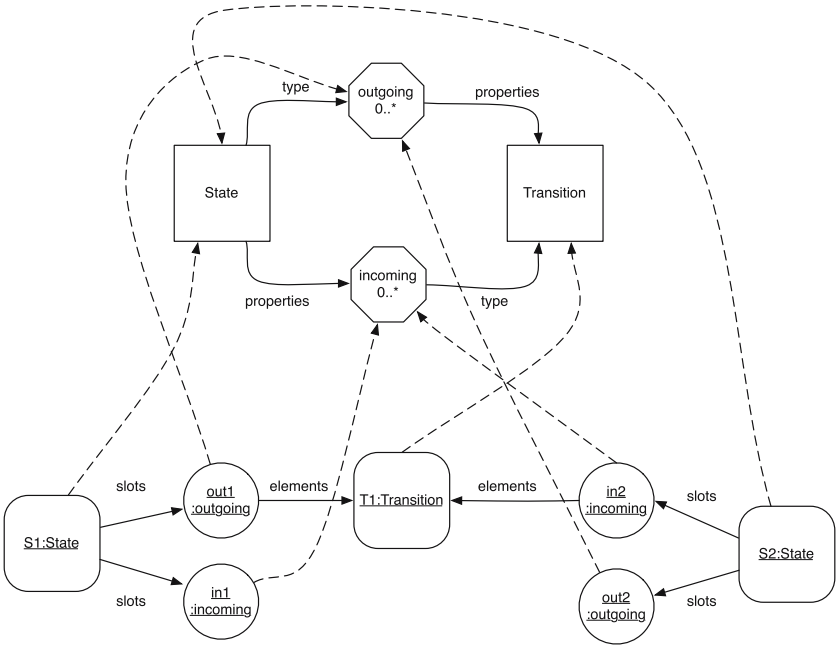

Fig. 7 Conformance of a model to its modeling language

we consider the expressiveness of a framework to be in the various property characteristics; primitive values are fairly uninteresting. Nevertheless, we will give a brief description of how they can be added to the framework, but we will not consider them any further in this article.

We can add various classes that represent primitive data types to C. For example, we can say that $\mathbb{Z} \in C$ and $\mathbb{S} \in C$ denote the class of integers and the class of strings, respectively. Then, we add a partial function from elements to data values, value : $E \nrightarrow \mathbb{Z} \cup \mathbb{S}$. It maps an element to its primitive value if said element is of the correct class. For example, an element e such that $\text{class}(e) = \mathbb{S}$ can be mapped to a string value, and thus $\text{value}(e)$ returns a string. Modifying primitive values is done by modifying the function value.

Thus, our primitive values are also elements and can technically contain slots as well, referencing other elements. While this is not common in for example programming languages, we feel this arrangement to be conceptually easier as it avoids further constraints.

4 Property characteristics

In the previous section, we have studied how properties can be used to relate model elements together. In this section, we discuss how different property characteristics such as multiplicity or composition can be used to constrain even further how elements can be related via slots. We define the characteristics of a property as a tuple:

$$\text{characteristics}_B = (\text{lower}, \text{upper}, \text{ordered}, \text{composite}, \text{opposite})$$

This tuple describes additional features of properties using several functions:

- lower : $P \to \mathbb{Z}^{0+} \setminus \infty$ represents the lower multiplicity constraint of a property (0, 1, 2, ..., excluding infinity).
- upper : $P \to \mathbb{Z}^{+}$ represents the upper multiplicity constraint (1, 2, ..., ∞).
- composite : $P \to \mathbb{B}$ is true if a property denotes composition.
- ordered : $P \to \mathbb{B}$ is true if a property denotes an ordered collection of elements.
- opposite : $P \to P \cup \{\Omega\}$ denotes the optional opposite of a property in a relation between two classes.

The rest of this section explains the semantics of these functions and how they effect several constraints on models.

4.1 Multiplicities

One of the simpler but more important concepts is the multiplicity constraint. The lower and upper characteristics constrain the amount of elements that can be referenced by a slot:

Model Constraint 4 *Valid amount of elements in a slot*: $(\forall s \in S \cdot \text{lower}(\text{property}(s)) \leq \#s \leq \text{upper}(\text{property}(s)))$

Since the amount of elements in a slot is bounded by the multiplicity characteristics of its property, the lower value should be less than the upper value. Otherwise, the multiplicity constraint cannot be satisfied by any slot:

Metamodel Constraint 1 *Property Multiplicity*: $(\forall p \in P \cdot \text{lower}(p) \leq \text{upper}(p))$

Multiplicities are used extensively in the UML and MOF language. These languages support the concepts of

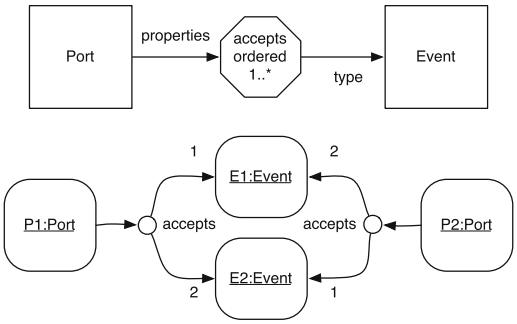

Fig. 8 *Top* A metamodel using the ordered property characteristic. *Bottom* A model conforming to this metamodel

multiplicity ranges, and the valid amount of elements in a slot is a subset M of $\mathbb{Z}^{0+}$. In practice, UML and MOF describe a multiplicity constraint as a set I of intervals (l, u) such that $M = \{x \cdot (\exists (l, u) \in I \cdot l \leq x \leq u)\}$.

4.2 Ordering

The ordering characteristic is used to model ordered collections of elements. An example of the usage of an ordered property is the parameters in a method of a class. Another more interesting example is shown in Fig. 8. The top of the figure shows a metamodel for a modeling language for the ports of an active event-based component. A port accepts a number of events and this is modeled using an ordered property. The bottom of the figure shows an example model where ports *P1* and *P2* accept events *E1* and *E2*. However, port *P1* considers that event *E1* has priority with respect to event *E2*, while *P2* gives priority to event *E2*.

This example remarks the fact that the ordering characteristic does not define an ordering of elements but an ordering of the elements referenced by one particular slot. We should also note that the ordering as such does not introduce new constraints in a model, although ordering should be taken into consideration in all the model constraints.

4.3 Bidirectionality

We have seen that a property can be used to define a UML or MOF 2 relation that is navigable by only one of its participants. However, we can also define bidirectional relations, by defining the opposite of a property.

Formally, the characteristic opposite : $P \rightarrow P \cup \{\Omega\}$ is a function that yields the opposite of a property, or the special constant Ω, which means that no opposite is defined. At the metamodel layer, we require that a property has itself as the opposite property of its opposite (iff it exists):

Metamodel Constraint 2 *Opposite properties*: $(\forall p \in P \cdot \text{opposite}(p) \neq \Omega \implies p = \text{opposite}(\text{opposite}(p))$

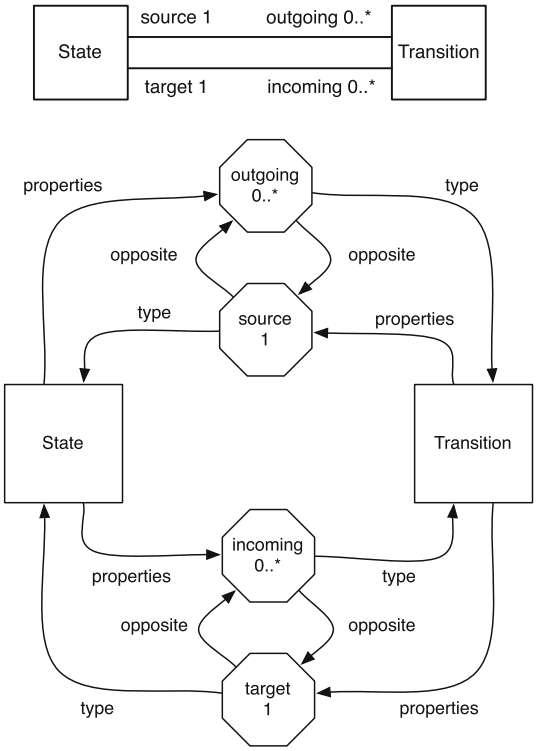

Fig. 9 *Top* Metamodel for a Statechart using navigable relations. *Bottom* The same metamodel as a graph of classes and properties

Figure 9 depicts a reviewed metamodel for a statechart. In the example, each property has another property as its opposite: *source* and *outgoing* are opposites and form a relation, as well as *target* and *incoming*. In a model, this relation means that when a *State s* has a *Transition t* in its *outgoing* slot, the *Transition t* will have *State s* in its *source* slot. In Fig. 10, state *S1* refers to transitions *T1* and *T2* in its *outgoing* slot, where the transitions refer to *S1* in their *source* slot.

At the model layer, we need to reflect that the contents of two opposite slots always refer to each other. This is captured in the following constraint for opposite slots:

Model Constraint 5 *Bidirectionality of slots*: $(\forall s \in S \cdot \text{opposite}(\text{property}(s)) \neq \Omega \implies (\forall e' \in \text{contents}(s) \cdot (\exists! s' \in S \cdot \text{slotOwner}(s') = e' \wedge \text{opposite}(\text{property}(s')) = \text{property}(s) \wedge \text{slotOwner}(s) \in \text{contents}(s'))))$

Our interpretation of relations is also shared by other authors, including Génova et al. [22]. However, according to some researchers, bidirectionality of two properties does not imply a bidirectionality requirement at the model layer. That is, model constraint 5 does not need to hold.

To see why this belief does not lead to a useful concept in a modeling language, consider Fig. 11. It is a valid model according to the statemachine metamodel presented earlier in Fig. 9, except for the fact that model constraint 5 does not hold. There are two cases where the constraint does not hold.

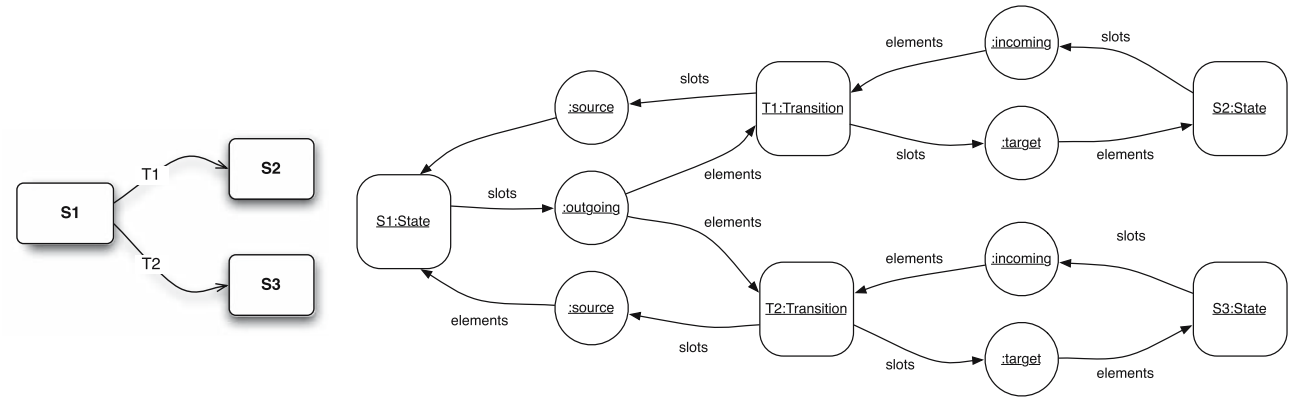

Fig. 10 *Left* Example Statechart. *Right* Statechart represented as *elements* and *slots*, conforming to the metamodel from Fig. 9

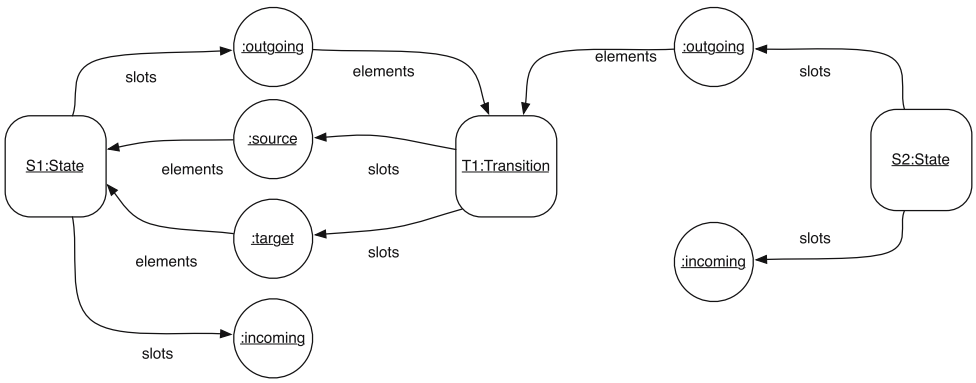

Fig. 11 A statemachine model without bidirectional slots. In the example T1 is an outgoing transition of S2, but T1 source state is S1

First, the *outgoing* connection between *S2* and *T1* does not have a *source* connection from *T1* to *S2*. Second, the *target* connection from *T1* to *S1* similarly does not have an *incoming* connection from *S1* to *T1*.

We firmly believe this example is nonsensical. If for some reason this is the intended interpretation of a modeling language, the metamodel should reflect it, as shown in Fig. 12. In this new metamodel, the properties source and outgoing are not opposites, and therefore, there is no constraint between their slots.

4.4 Composition

A very important property characteristic is composition. Composition is used to denote hierarchy and ownership in a model. It is a very important concept that aids us in organizing models as a collection of smaller parts. A composition property imposes a rather restrictive constraint over its slots: an element can only be referenced by one composition slot at a time and it should not be possible to create cyclic compositions.

Figure 13 shows an example of the use of composition in a metamodel. The top of the figure contains a simplified metamodel, in both UML and our notation, for a class modeling language containing a package and a class. A package may contain classes and each class may contain inner classes. However, a class should not be directly owned by both a package and by a class simultaneously, and neither can an inner class directly or transitively be an inner class of itself. The bottom of the figure contains a model that presents these two cases: class *C1* is owned by package *P1* and class *C3* simultaneously and there is a composition cycle between classes *C1*, *C2* and *C3*. Therefore the model at the bottom of the figure does not conform to the metamodel at the top of the figure.

Formally, we say that an element x is the owner or *parent* of an element e if e is referenced by a composite slot s of x. We define the function parent : $E \rightarrow \mathcal{P}(E)$ to return either the empty set if no parent for an element exists, or a set consisting of all the parent elements. Thus, the size of this set should be at most one.

$$\begin{aligned}\text{parent}(e) = \{x \;\cdot\; x \in E \wedge (\exists s \in S \;\cdot\; \text{slotOwner}(s) = x \\ \wedge\, \text{composite}(\text{property}(s)) \wedge e \in \text{contents}(s))\}\end{aligned}$$

Thus, an element cannot be owned by the same element via two different composite slots:

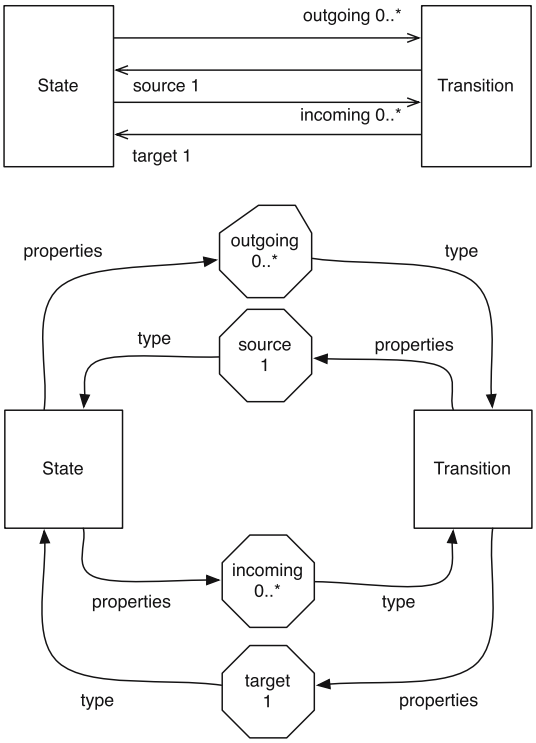

Fig. 12 *Top* Metamodel for a Statechart using non-bidirectional relations. *Bottom* The same metamodel as a graph of classes and properties

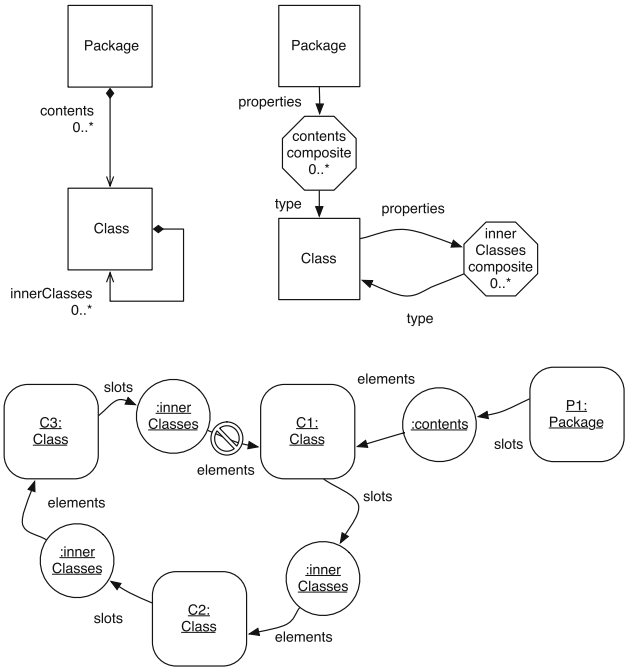

Fig. 13 Example of composition. The model at the *bottom* does not conform to the metamodel at the *top*

Model Constraint 6 *Only in one composite slot*: $(\forall e \in E \cdot \neg(\exists s_1, s_2 \cdot \text{slotOwner}(s_1) = \text{slotOwner}(s_2) \wedge \text{composite}(\text{property}(s_1)) \wedge \text{composite}(\text{property}(s_2)) \wedge e \in \text{contents}(s_1) \wedge e \in \text{contents}(s_2))$

As stated previously, an element cannot be the owner of itself, directly or transitively:

Model Constraint 7 *Composition is acyclic*: $(\forall e_1, \ldots, e_n, e_{n+1} \in E \cdot (\forall i \cdot 1 \leq i \leq n \implies e_i \in \text{parent}(e_{i+1})) \implies e_1 \neq e_{n+1}), \quad \forall n \geq 1$

This also implies that a relation at the metamodel level cannot be made from two composite properties, since such slots would be void.

Metamodel Constraint 3 *Both properties in a relation cannot be composite*: $(\forall p \in P \cdot \text{composite}(p) \wedge \text{opposite}(p) \neq \Omega \implies \neg\text{composite}(\text{opposite}(p)))$

Defining metamodels using both composition and multiplicity range characteristics has an interesting consequence. It is possible to declare a chain of several classes such that it is mandatory for an instance of a class to own at least one instance of the next class, et cetera, until a cycle is created. Thereby all classes would be void, since only an infinite chain of elements could conform to them. We can prohibit this with the following constraint:

Metamodel Constraint 4 *No infinite chain of compositions*: $(\forall c_1, \ldots, c_n, c_{n+1} \in C \cdot (\forall i \cdot 1 \leq i \leq n \implies (\exists p \in \text{effectiveProperties}(c_i) \cdot \text{composite}(p) \wedge \text{owner}(p) = c_i \wedge c_{i+1} = \text{type}(p) \wedge \text{lower}(p) \geq 1)) \implies c_1 \neq c_{n+1}), \quad \forall n \geq 1$

Composition is used extensively in the definition of UML and MOF and it also appears in the Graph eXchange Language [55]. Its semantics in the context of UML has been studied by Barbier et al. [12,25]. Composition brings many advantages when building tools that need to traverse or transform models. If we only take slots of a composite property and elements into account, the resulting graph forms a tree (or a forest). This allows us to use efficient traversal algorithms. Also, this arrangement maps well to the XMI, since an XML document has a tree structure.

In this paper, we use the concept of *strict composition* or "black diamonds" as described in [25]. Another alternative interpretation of composition is *shared composition*. In this case, the composition links should be acyclic, but an element can have more than one parent. To achieve this interpretation of a modeling language, model constraint 6 should be removed. The resulting graph forms a directed acyclic graph.

4.5 Attributes

We should note that our metamodeling language does not have any special provision to model attributes such as in MOF or EMF. This is due to the fact that we can model an attribute by using a combination of the property characteristics

that we have already defined. In our approach, we consider an attribute definition equivalent to a property that is a composition and does not have an opposite. This reduces the amount of defined concepts at the metamodeling layer and thus simplifies the structure of metamodels and models. We have validated this idea in our modeling tool and found no problems.

5 Class specialization

In this section, we introduce the concept of class specialization as a mechanism to organize and simplify large metamodels. Class specialization is the same concept as class inheritance in OO programming languages. A class can be a specialization of one or more base classes and as a consequence it inherits all the properties of its base classes. Without class specialization, the definition of UML would require many additional properties and model transformations would be more complex and cumbersome to create and maintain. Therefore, class specialization is used extensively in the definition of UML and MOF.

As an example, the UML 1.x metamodels use class specialization to model state hierarchy. Figure 14 shows a simplified Statechart model where we specialize the *State* class into *CompositeState*. In UML, class specialization is represented diagrammatically as an edge between the base class and the specialized class with a triangular arrow head pointing to the base class. In our example, a *CompositeState* inherits all the properties of a normal state but adds an additional property to model substates. A substate can be a *State* or a *CompositeState*.

We should also note that a metamodeling language should support multiple inheritance since it is used extensively in MOF. This has already been noticed by for example Kleppe [29]. In order to formalize class specializations we will need to extend our definition of a metamodel by adding the concept of generalizations of a class. A modeling language supporting class specialization is then defined by the tuple:

$$ML_G{=}(C, \text{generalizations}, P, \text{owner}, \text{type}, \text{characteristics})$$

We define the generalizations of a class with the function $\text{generalizations} : C \rightarrow \mathcal{P}(C)$. We denote by $\subseteq_c$ the extended generalization between classes that is defined as the reflexive transitive closure of the generalization relation: $\subseteq_c = \{(c, d) \cdot d \in \text{generalizations}(c)\}^*$. Intuitively, given two different classes c and d, we say that c is a subclass of d iff $c \subseteq_c d$. We also note that $\text{characteristics}_G = \text{characteristics}_B$, since no new property characteristics need to be added.

We should now review the concept of effective properties of a class for a metamodeling language supporting class specialization. The effective properties of a class shall include all the properties owned directly by that class and all the effective properties of its superclasses:

$$\text{effectiveProperties}_G(c) = \text{properties}(c) \cup \bigcup\{\text{effectiveProperties}_G(d) \cdot d \in \text{generalizations}(c)\}$$

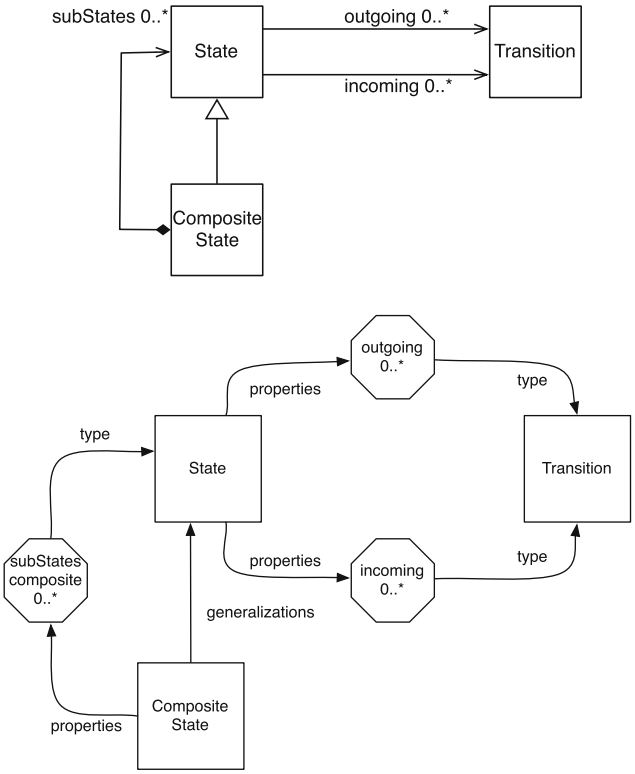

Fig. 14 A metamodel using class specialization

We should also review what the effective types of a property in a language supporting class specialization are. In this language, slots are covariant: a slot may contain elements whose class is the basic type of its property or any subclass of that type.

$$\text{effectiveType}_G(p) = \{c \in C \cdot c \subseteq_c \text{type}(p)\}$$

We can see an example of these definitions in the metamodel shown in Fig. 14. From it, we can see that:

$$\text{effectiveProperties}_G(\textit{CompositeState}) = \{\textit{outgoing}, \textit{incoming}, \textit{subStates}\}$$

That is, the effective properties of *CompositeState* are the two properties owned by *State* and the additional property directly owned by *CompositeState*. Similarly, we have:

$$\text{effectiveType}_G(\textit{subStates}) = \{\textit{State}, \textit{CompositeState}\}$$

An example model using this metamodel can be seen in Fig. 15.

Because the effective properties of a class are defined by itself and its transitive superclasses, we require the gener-

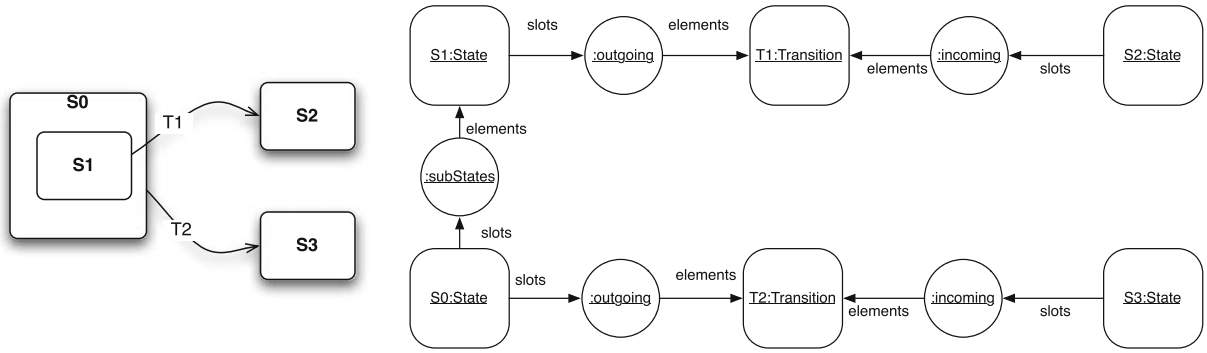

Fig. 15 Example model using specialization. A CompositeState is a specialization of State

alization relation to be acyclic. Otherwise all classes in a generalization hierarchy would have the same set of effective properties and they would for all practical purposes be indistinguishable from each other. This leads to the following metamodel constraint:

Metamodel Constraint 5 *Generalization is acyclic*: $\neg(\exists e \in C \cdot (e, e) \in \{(c, d) \cdot d \in \text{generalizations}(c)\}^+)$

6 Property subsetting

In this section, we introduce the concept of property subsetting to our metamodeling language, one of the most intriguing concepts introduced in MOF 2.0 and the UML 2.0 Infrastructure. Property subsetting allows us to specialize an existing property into a new property with a different basic type and different characteristics, while still retaining the old, existing property. The intuition is that the specialized property is a subset of the original property. Therefore, elements in a slot of a subset property should also be included in the slot of the original property.

As an example, we present yet another version of a simplified metamodel for UML class diagrams in Fig. 16. We first provide a general concept of a container and its children elements using the *Namespace* and *Element* classes. Each *Namespace* element has a slot named *ownedElement* representing its contents.

Then we specialize *Namespace* into a *Class* and add two *subset properties* called *ownedAttribute* and *ownedOperation* to keep attributes and operations. These properties are *subsetting* the *ownedElement* property. We also add two subset properties of *ownedAttribute* to *Class*, called *ownedPublicAttribute* and *ownedPrivateAttribute*. This example also shows how a subset and a union property may be in the same class.

We introduce a new property characteristic to our metamodeling language in order to support subset properties:

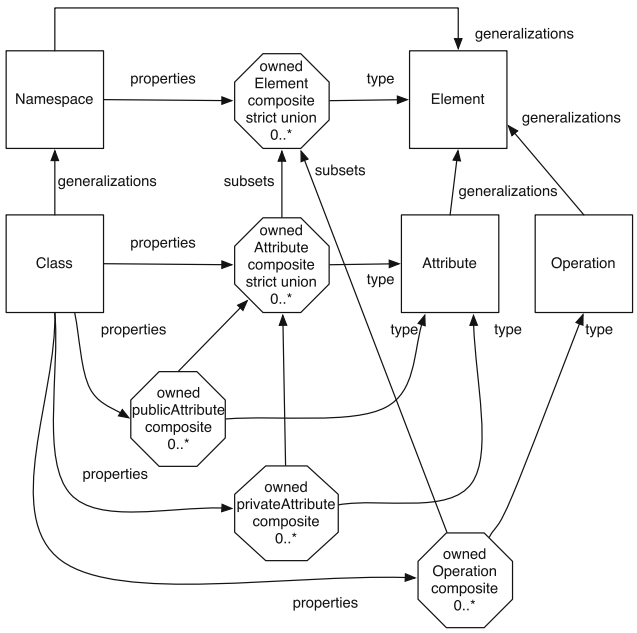

Fig. 16 Example metamodel for UML Class diagrams using subset properties

– supersets : $P \rightarrow \mathcal{P}(P)$ represents the properties of which a property is a subset.

We will also introduce the characteristic strictUnion later. Thereby, we can define a modeling language supporting property subsetting as the tuple:

$$ML_S = (C, \text{generalizations}, P, \text{owner}, \text{type}, \text{characteristics}_S)$$
$$\text{characteristics}_S = (\text{lower}, \text{upper}, \text{ordered}, \text{composite}, \text{opposite}, \text{supersets}, \text{strictUnion})$$

We denote subsetting between properties by the $\subseteq_p$ relation, i.e., $\subseteq_p = \{(p, q) \cdot q \in \text{supersets}(p)\}^*$. The relation $\subseteq_p$ is a partial order on P.

In a model, we say that a slot r is a subset of another slot s if $\text{property}(r) \subseteq_p \text{property}(s)$ and they have the same owner.

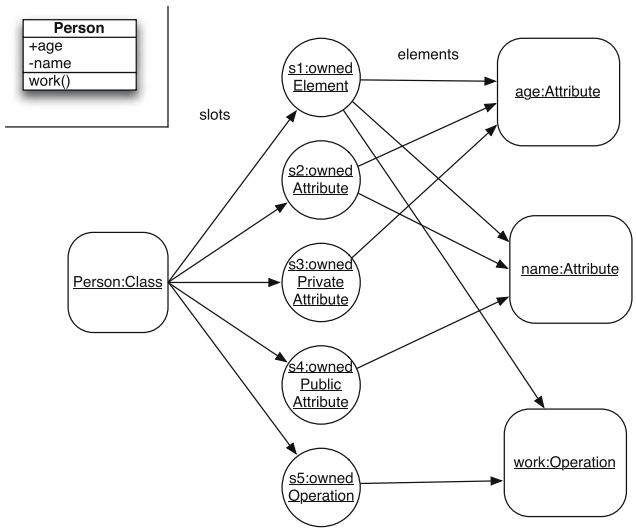

Fig. 17 Example model based on the metamodel shown in Fig. 16

The slot subsetting relation is thus defined by:

$$\subseteq_s = \{(r, s) \cdot \text{slotOwner}(r) = \text{slotOwner}(s) \wedge \text{property}(r) \subseteq_p \text{property}(s)\}^*$$

It can be split into several partial orders, one for each slot and its super- and subsets.

The contents of a slot r subsetting another slot s must be a subset of the contents of s. Also, MOF [41, p. 56] tells us that *"The slot's values are a subset of those for each slot it subsets."*. In the case of unordered slots, this is formalized using the following constraint:

Model Constraint 8 *Unordered slots*: $(\forall r, s \in S \cdot r \subseteq_s s \wedge \neg\text{ordered}(\text{property}(s)) \implies \text{contents}(r) \subseteq \text{contents}(s))$

We can see an example model based on subset properties in Fig. 17. The model represents a UML class with one public attribute, one private attribute and one operation. The element representing the class has five different slots and the elements referenced in each slot is constrained by the subset properties. In the example, we have $s_5 \subseteq_s s_1 \wedge s_4 \subseteq_s s_2 \wedge s_3 \subseteq_s s_2 \wedge s_2 \subseteq_s s_1$.

6.1 Subsets and ordering

For ordered slots we also wish to preserve order. That is, when elements occur in a specific order in r, they should occur in the same order in s, although s might contain more elements in between.

Model Constraint 9 *Ordered slots*: $(\forall x, y \in E, r, s \in S \cdot x \in \text{contents}(r) \wedge y \in \text{contents}(r) \wedge x \preceq_r y \wedge r \subseteq_s s \wedge \text{ordered}(\text{property}(s)) \implies x \in \text{contents}(s) \wedge y \in \text{contents}(s) \wedge x \preceq_s y)$

6.2 Union properties

A property is called a union property if it has one or more properties that subset it. In our framework, it is not necessary to declare a property as a union, since a designer of a metamodel cannot know in advance if a new subset property will be defined in the future, possibly in some other metamodel.

6.3 Strict unions

The UML 2.0 Infrastructure also introduced the concept of strict union. The standard states on p. 126 that *"This means that the collection of values denoted by the property in some context is derived by being the strict union of all of the values denoted, in the same context, by properties defined to subset it. If the property has a multiplicity upper bound of* 1*, then this means that the values of all the subsets must be null or the same."*. In other words, a derived union property can be seen as the strict union of its subsets. A slot with a property that is a strict union cannot contain elements that do not appear in any of its subsets.

We introduce a new property characteristic to our metamodeling language in order to support strict unions:

- strictUnion : $P \to \mathbb{B}$ is true if a property is a strict union.

The UML uses the qualifier "{union}" to denote a property as a strict union. Since all our properties are unions, we use the qualifier "{strict union}" to avoid confusion. In the example, the contents of *ownedElement* and *ownedAttribute* slots are strict unions of the contents of the subsetting slots.

The concept of strict unions implies that a new model constraint needs to be defined:

Model Constraint 10 *Elements in a strict union*: $(\forall s \in S \cdot \text{strictUnion}(\text{property}(s)) \implies \text{contents}(s) = \bigcup\{\text{contents}(r) \cdot r \prec_s s\}$

6.4 Subsets and substitutability

The rationale for the proposed model constraints is to allow type substitutability in model transformations, queries and code generators. A specialized class has the same properties with the same characteristics as its base classes, while containing new definitions to specialize these properties. Thus, subsetting preserves Liskov substitutability.

As an example, Fig. 18 shows a model based on the example metamodel for circuits shown in Fig. 2. Here, a given transistor can be seen as a generic component with a number of pins or as an element of type *Transistor* that has three specific pins. The benefit is that it is possible to define algorithms and transformations which work on the base abstract circuit; that is, only considering wires, pins and components,

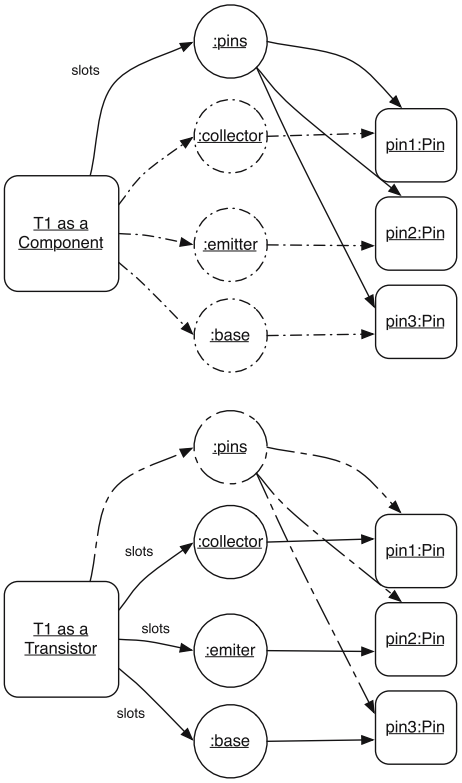

Fig. 18 Example of a model: *top* as interpreted according to the base language. *Bottom* the extended language

without caring for the details, whereas algorithms that are targeted to specific components can rely on a more refined abstract syntax.

In the rest of this section, we review how the concept of subset properties interacts with the other concepts in our metamodeling language.

6.5 Subsets and multiplicities

It can easily be seen that a property subsetting another property should have a lower (or the same) upper limit than the other property. This can be formalized with the following metamodel constraints:

Metamodel Constraint 6 *Upper multiplicity in subset properties*: $(\forall p \in P \cdot (\forall q \in \text{supersets}(p) \cdot \text{upper}(p) \leq \text{upper}(q)))$

The justification for this constraint can be shown with slots r and s such that $r \subset_s s$, $\text{property}(r) = p$, $\text{property}(s) = q$ and $\text{upper}(p) > \text{upper}(q)$, and by filling the slot r with elements so that $\#r = \text{upper}(p)$. Then $\#s \geq \#r = \text{upper}(p) > \text{upper}(q) \implies \#s > \text{upper}(q)$, which violates the upper limit of q.

Property subsetting does not raise any new restrictions on the lower limits of the properties. This is because more elements can always be inserted into a slot until its size is at least that of the greatest lowest limit in any transitive sub- or superset. Thus, model constraint 4 is sufficient for the lower multiplicity constraints.

6.6 Subsets and class specialization

A property should only subset another property if the effective types of it are a subset of the effective types of the other. The same remark is also stated in the UML 2.0 Infrastructure [42, p. 125]: *A property may be marked as a subset of another, as long as every element in the context of the subsetting property conforms to the corresponding element in the context of the subsetted property.*

The left side of Fig. 19 shows a (nonsensical) metamodel, in which property d of class C subsets property b of class A. The type of b is class B and the type of d is class D; however, D is not a subclass of B.

We should explore how the existing constraints affect the elements in the slots of a model based on the metamodel in the left side of Fig. 19. Since the elements in a slot s_d conforming to property d should be of type D, the elements in slot s_b conforming to property b should be of type B and the elements in s_d should also be in s_b, the slot s_d cannot have any elements. This is shown in the following derivation, based on the structured derivation approach in [11]:

$$
\begin{aligned}
& s_d, s_b \in S \land \text{property}(s_d) = d \land \text{property}(s_b) = b \\
& \land s_d \subseteq_s s_b \land \text{effectiveType}(d) = \{D\} \\
& \land \text{effectiveType}(b) = \{B\} \\
\implies & \{\text{introduce class of elements in slot, unordered} \\
& \ \text{slot constraints, simplify}\} \\
& (\forall e \in \text{contents}(s_d) \cdot \text{class}(e) \in \{D\}) \land \\
& (\forall e' \in \text{contents}(s_b) \cdot \text{class}(e') \in \{B\}) \land \\
& (\text{contents}(s_d) \subseteq \text{contents}(s_b)) \\
\implies & \{\text{definition of subset, membership in a singleton set}\} \\
& (\forall e \in \text{contents}(s_d) \cdot \text{class}(e) = D) \land \\
& (\forall e' \in \text{contents}(s_b) \cdot \text{class}(e') = B) \land \\
& (\forall e'' \in \text{contents}(s_d) \cdot e'' \in \text{contents}(s_b)) \\
\implies & \{\text{elements in } d \text{ should also satisfy the constraint for } b\} \\
& (\forall e \in \text{contents}(s_d) \cdot \text{class}(e) = D \land \text{class}(e) = B) \\
\implies & \{D \neq B\} \\
& \text{contents}(s_d) = \emptyset
\end{aligned}
$$

Based on this discussion, we consider an additional constraint over a metamodel: the fact that a property can subset another property only from the reflexive transitive superclass closure of its owner:

Metamodel Constraint 7 *Subset only from owner or its superclasses* $(\forall p, q \in P \cdot p \subseteq_p q \implies \text{owner}(p) \subseteq_c \text{owner}(q))$.

However, this restriction is not strong enough. It would still be possible to cyclically subset a property within the

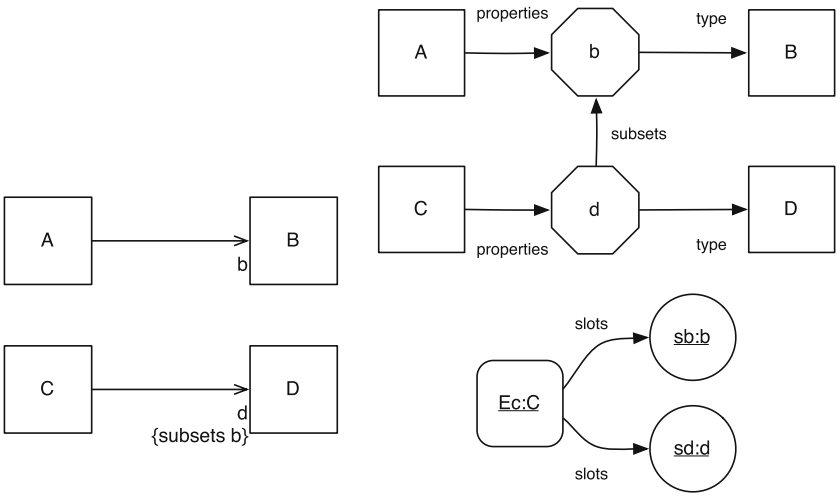

Fig. 19 Example of the interaction of subsets and subtyping: property *d* is void

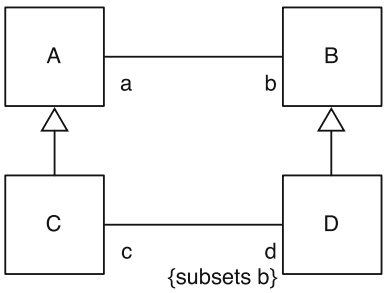

Fig. 20 Example of the interaction of subsets and opposite properties

same class. This is not a useful construct since any slots of the properties in the cycle would consist of the exact same elements. Thus, property subsetting must be acyclic:

Metamodel Constraint 8 *The property superset relation is acyclic*: $\neg(\exists e \in P \cdot (e, e) \in \{(p, q) \cdot q \in \text{supersets}(p)\}^+)$

It can be noted that multiple inheritance forms very complicated inheritance hierarchies, among them the *diamond inheritance* structure. This leads to a possibility where property subsetting also has a diamond (or even more complicated) structure.

6.7 Subsets and opposite properties

We should now study the possible interactions between subset and opposite properties. Let us consider the metamodel in Fig. 20. In this metamodel, the subset property *d* has an opposite property *c* which is not a subset.

Let us assume that there is a model with two elements e_c and e_d conforming to classes *C* and *D*, respectively. According to the metamodel, element e_c has two slots that we name s_d and s_b conforming to properties *d* and *b*, respectively. Similarly, element e_d has two slots s_c and s_a. We wish to add element e_d to the slot s_d. This is shown in bold in the right side of Fig. 20. Since the property *d* has an opposite, we also need to add e_c to the slot s_c of e_d in order to satisfy model constraint 5 regarding bidirectional slots. Since $d \subset_s b$, we also need to include e_d in the slot s_b to satisfy model constraint 8 about unordered subset slots. Finally, since property *b* has an opposite property named *a*, we should include e_d in the elements of s_a. Thus, e_d is in s_a and in s_c, and the net effect is as if *c* were a subset of *a* anyway, even though that was not stated in the metamodel.

The conclusion is that we claim that *c* needs to subset *a*, if for nothing else than documentation purposes. There are several faults in MOF 2.0 and UML 2.0 where this rule is violated. Fortunately, the correction is simple by saying that (in our example) *c* needs to subset *a*. In any case, this example emphasizes the need for the following constraint:

Metamodel Constraint 9 *The opposite of a subset property must be a subset*: $(\forall p, q \in P \cdot p \subseteq_p q \wedge \text{opposite}(p) \neq \Omega \implies \text{opposite}(p) \subseteq_p \text{opposite}(q))$

6.8 Subsets and composition

We need to redefine the composition constraints to take into account the subset properties. Due to the subset constraints, an element may be in more than one composition slot at a given time, as long as these slots are not independent. This replaces model constraint 6:

Model Constraint 11 (*Subset*) *Only in one composite slot*: $(\forall e \in E \cdot \neg(\exists s_1, s_2 \cdot \text{slotOwner}(s_1) = \text{slotOwner}(s_2) \wedge (\text{property}(s_1) \,||_p\, \text{property}(s_2)) \wedge \text{composite}(\text{property}(s_1)) \wedge \text{composite}(\text{property}(s_2)) \wedge e \in \text{contents}(s_1) \wedge e \in \text{contents}(s_2))$

In Fig. 21, we see different cases with composite and non-composite properties. Cases (1) and (2) are legal and quite self-explanatory: in the first case, all *A* and *C* elements own their *B* and *D* elements via the *a*–*b* relation, and in the second case the *c*–*d* relation can be used to own some of the

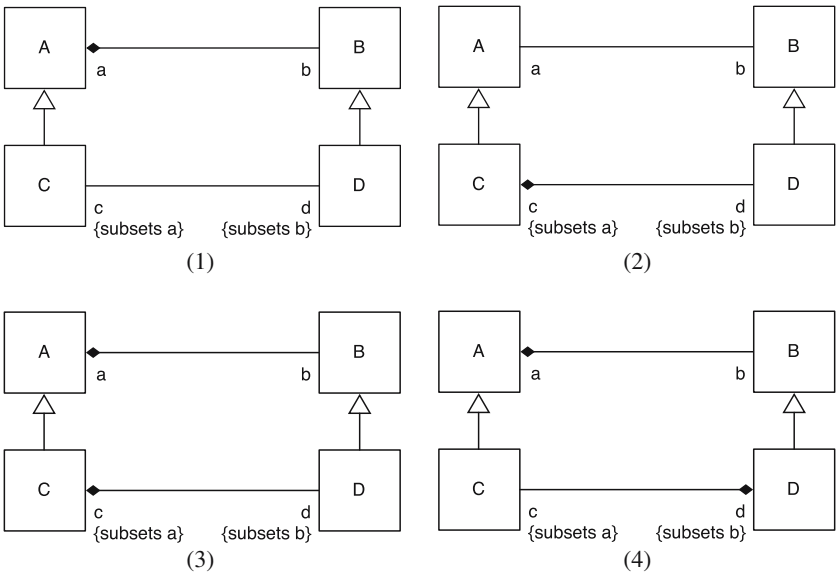

Fig. 21 Subsetting with composite and noncomposite properties

D elements and the rest can be referenced via only the *a–b* relation, and can thus be owned by some other element.

Case (3) can be considered legal by discounting the composition at the *c–d* relation without any loss in information, since any elements owned via the *c–d* relation must also be owned via the *a–b* relation. This is what model constraint 11 allows. Case (4) is void since any elements of types *C* and *D* that are connected at the *c–d* relation are also connected at the *a–b* relation, thereby creating a cyclic composition and therefore violating a model constraint. Thereby the following metamodel constraint should be defined:

Metamodel Constraint 10 *No circular transitive composition with subsets*: $(\forall p \in P \cdot \text{composite}(p) \implies \neg(\exists q \in P \cdot \text{opposite}(q) \neq \Omega \wedge p \subset_p q \wedge \text{composite}(\text{opposite}(q))))$

We can find examples of the three first cases in UML 2.0, all in Fig. 11.5 of [42, p. 109]. Case (1) can be found in the *association–memberEnd* relation, case (2) is found in the *owningAssociation–ownedEnd* relation and case (3) in the *class–ownedAttribute* relation.

7 Alternative language extension mechanisms

In this section, we briefly go through various additional language extension mechanisms. We also provide a motivation on why we do not include them in our framework.

7.1 Covariant specialization

Covariant specialization is similar to subsetting in that it relates two relations at the metamodel layer. However the semantics are different. In a covariant environment, the specialized relation cannot be modified for elements which are instances of the subclasses.

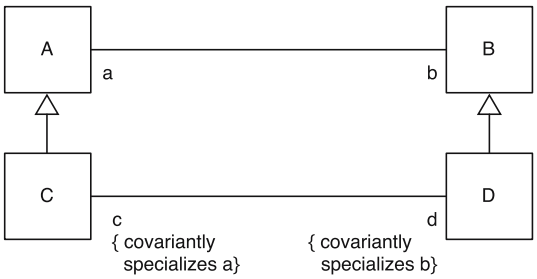

Fig. 22 Notation for covariant specialization

As an example of covariant specialization, let us assume that e_c is an element of type *C* shown in Fig. 22. It is not possible to insert elements of type *B* into the *b* slot of e_c, only elements of type *D* into the *d* slot. The *c–d* relation is a *covariant specialization* of the *a–b* relation. The *a–b* relation has been rendered obsolete (or at least read-only) in the context of element e_c.

Covariance is a subject that often comes up in the semantics of methods of OO programming. Function parameter type contravariance and return type covariance are rather inconvenient in practical situations and thus a type-unsafe function parameter type covariance is used for specialization. A similar argument also holds for element slots. Property subsetting aims to provide a new way to represent relationships between elements. It must nevertheless be noted that as Giuseppe Castagna has asserted, there are uses for a covariant environment when compared with a contravariant or invariant environment. Thus subsetting and covariance are not opposing but complementing constructs in OO programming and thus in modeling [17].

The major difference between covariance and subsetting is that in a covariant environment, substituting an element of a specific type with an element which is a covariant specialization of that type can result in programs no longer working. Thereby covariant specialization breaks Liskov substitutability and that is the reason we do not include it in our framework, although we realize it is an important concept. On the other hand, subsetting allows the slots defined by the properties in a superclass to be used in an instance of a subclass.

We note that contrary to subsetting, it might be necessary for the metamodel developer to explicitly declare the possibility of covariant specialization, instead of leaving the decision open for the future. This is because metamodel users need to realize that slots conforming to covariantly specialized properties may not be available in the future when their algorithm or function receives instances of the subclasses. That is, in the example it might be necessary to state a priori that the *a*–*b* relation can be covariantly specialized, as a warning mechanism for users and tools. Subsetting can be declared when required, whereas covariant specialization must be planned beforehand.

7.2 Property redefinition

MOF 2.0 introduces the concept of property redefinition. It is our understanding that a property redefinition is an arbitrary replacement of the characteristics of a property in a subclass that overrides the subsetted property and renders it unusable in the subclass.

We can formalize this concept in our framework by introducing a new property characteristic redefines : $P \rightarrow \mathcal{P}(P)$, and by defining the set of effective properties of a class as follows:

$$\begin{aligned}&\text{effectiveProperties}(c) = \\ &\quad \bigcup\{\text{properties}(s) \cdot c \subseteq_c s\} \\ &\quad \setminus \bigcup\{\text{redefines}(p) \cdot p \in \text{properties}(s) \wedge c \subseteq_c s\}\end{aligned}$$

Usually, the redefining property has the same name as the redefined property, and exists in a subclass of the class of the redefined property. In data modeling terms, this means that programs will obtain the redefining property when they use an element of the subclass type.

However, we should note that there are no constraints between a property and its redefinition. A redefinition could be covariant or contravariant in some characteristics, and otherwise compatible or incompatible in others with respect to its redefined property. Using property redefinitions in a language extension breaks Liskov substitutability and therefore transformations and tools based on the original language. Therefore, we consider that property redefinition is not a safe construct and it should not be included in a metamodeling language.

8 Related work

Several other researchers have formalized metamodeling languages and model layers. For example, Thomas Baar has defined the CINV language [10] using a set-theoretic approach, but our approach is more general in that we also support generalizations. The benefits of a set-theoretic approach is that it avoids a metacircularity whereby one (partially) needs to understand the language to be able to learn the language. Álvarez et al. [7] describe such a static OO metacircular modeling language, and the Metamodeling Language Calculus [18] by Clark et al. is another very sophisticated one. Nytun et al. present in [34] a modeling framework in which all model layers are represented uniformly.

Akehurst et al. present in [19] the structure of a metamodel and its semantics using OCL. Our rationale for not using OCL to define the model and metamodel constraints is that the definition of the navigation in OCL expressions actually depends on the metamodeling framework.

More recently, Jouault and Bézivin have presented KM3 [27], a metamodeling language targeted towards domain-specific modeling languages. This is one of the most influential works for this article since the notion of model conformance presented here is based on it.

However, the main contribution of this article comes from the definitions of property subsets in a language with multiple inheritance, which neither metamodeling nor traditional OO language descriptions explain. Several authors use relation inheritance without defining exact semantics, and some say that it denotes covariance. An example of this covariant specialization is the multilevel metamodeling technique called VPM by Varro and Pataricza [53], which also limits itself to single inheritance. We argue that property subsetting is not the same concept as covariant specialization, and requires different semantics.

Carsten Amelunxen, Tobias Rötschke and Andy Schürr are authors of the MOFLON tool [8] inside the Fujaba framework [33]. MOFLON claims to support subsetting, but no description of the formal semantics being used is included. It is not clear if their tool works in the context of subsets between ordered slots, or with diamond inheritance with subsetting. Markus Scheidgen presents an interesting discussion of the semantics of subsets in the context of creating an implementation of MOF 2.0 in [46]. To our knowledge, this has so far been the most thorough attempt to formalize subsetting.

The OO and database research communities are also researching a similar topic, although it is called relationship or association inheritance, or first-class relationships. In [15], Bierman and Wren present a simplified Java language with first-class relationships. In contrast with our work, they do not support multiple inheritance, bidirectionality or ordered properties; all of these constructs are common in modeling and in the UML 2.0 specification. However, relationship

links are explicitly represented as instances, and they can have additional data fields (just like the AssociationClass of UML). As the authors have noticed, the semantics of link insertion and deletion is not without problems.

Albano et al. present in [6] a relationship mechanism for a strongly typed OO database programming language. It also handles links as relationship instances, but without additional data fields. Multiple inheritance is supported, but ordered slot contents are not.

Finally, we should note there is an important ongoing discussion on the conceptual role of metamodeling and metamodeling languages in articles such as [9,23]. These works describe the conceptual relationship between different metamodeling levels or layers. Our work focuses on the concrete constraints between two specific levels and it clearly exhibits the two metadimensions described in [9], where every model element *logically conforms* to a given metamodel class while it is *physically represented* as an element.

9 Conclusions

In this article we have explored the main concepts used in a metamodeling approach that supports class specialization and property subsetting. We have achieved this by building the metamodeling framework from the ground up using successive set-theoretic definitions of the structural semantics. Each definition adds a concept to our modeling framework: multiplicities, bidirectionality, ordering, composition, class specialization, subsetting, unions and strict unions.

We have also briefly discussed other language extension mechanisms. We have argued that covariant specializations make classes nonsubstitutable, that arbitrary property redefinitions are not a safe extension mechanism and that package merges do not provide anything fundamentally new as they can be described in terms of previous mechanisms. Therefore, we have not included these concepts in our approach.

The contribution of this paper is important because it emphasizes the need for all new metamodel language concepts to form an integrated whole and because it defines the new property characteristics of subsets and unions from MOF 2.0. We realize that all new metamodeling constructs interact with all the old metamodeling constructs. We have to ensure that the semantics of all combinations of these constructs make sense by declaring suitable metamodel and model constraints.

The OMG modeling standards do not describe subset and union properties in detail, not even informally, and therefore they cannot be applied in practice. In this article, we have formalized a simple modeling framework that supports subsets and derived unions. It discusses the relevant model constraints that must be upheld by any valid model.

There are some limitations in the work presented in this article. The framework and especially subsetting as proposed is restricted to slots with unique elements. Slots where the same element can occur several times (bags) are not considered. Although bags can be defined in MOF 2.0, they are not used in the definition of the UML 2.0 Superstructure. It must also be stressed that we do not cover several important aspects of MOF 2.0, such as association end ownership or navigability.

We have implemented the metamodeling language defined in this article in our modeling tool Coral. Coral is open source and available at http://www.mde.abo.fi/. An important extension to our framework is the semantics of the operations that can be performed on models while satisfying the model constraints. These basic model operations over models containing subset and union properties are defined in [5].

Unfortunately, we know of no modeling tools that support subsets as extensively as discussed in this article. At the time of writing, the Eclipse EMF model repository does not implement subset properties, although the feature is planned. It is not clear what the semantics will be, though.

In conclusion, we consider that there is a need in the modeling community to standardize on one intuitive explanation and a rigorous formalization of subset properties and derived unions, so tools based on MOF 2.0 and UML 2.0 can be implemented and be interoperable. This article presents a proposal in this direction that we hope it can help other researchers and tool developers to define a common understanding for MOF 2.0 and UML 2.0.

Acknowledgments The authors would like to thank Patrick Sibelius for insightful discussions. Marcus Alanen would like to acknowledge the financial support of the Nokia Foundation.

Appendix A: A simple metamodeling language

This appendix lists a summary of the final structure of metamodels and models as well as their constraints.

The metamodels

The metamodels are defined by:

$$ML_S = (C, \text{generalizations}, P, \text{owner}, \text{type}, \text{characteristics})$$
$$\text{effectiveProperties}_S(c) = \text{properties}(c) \cup \bigcup\{\text{effectiveProperties}_S(d) \cdot d \in \text{generalizations}(c)\}$$
$$\text{effectiveType}_S(p) = \{c \in C \cdot c \subseteq_c \text{type}(p)\}$$

The characteristics of the properties, including subsets and strict unions, is defined by:

$$\text{characteristics}_S = (\text{lower}, \text{upper}, \text{ordered}, \text{composite}, \text{opposite}, \text{supersets}, \text{strictUnion})$$

such that:

- lower : $P \to \mathbb{Z}^{0+} \setminus \infty$ represents the lower multiplicity constraint of a property (0, 1, 2, . . ., excluding infinity).
- upper : $P \to \mathbb{Z}^{+}$ represents the upper multiplicity constraint (1, 2, . . . , ∞).
- composite : $P \to \mathbb{B}$ is true if a property denotes composition.
- ordered : $P \to \mathbb{B}$ is true if a property denotes an ordered collection of elements.
- opposite : $P \to P \cup \{\Omega\}$ denotes the optional opposite of a property in an association between two classes.
- supersets : $P \to \mathcal{P}(P)$ represents the set of properties of which a property is a subset.
- strictUnion : $P \to \mathbb{B}$ is true if a property is a strict union.

The metamodel constraints

Metamodel Constraint 1 *Property Multiplicity*: $(\forall p \in P \cdot \text{lower}(p) \leq \text{upper}(p))$

Metamodel Constraint 2 *Opposite properties*: $(\forall p \in P \cdot \text{opposite}(p) \neq \Omega \implies p = \text{opposite}(\text{opposite}(p))$

Metamodel Constraint 3 *Both properties in a relation cannot be composite*: $(\forall p \in P \cdot \text{composite}(p) \wedge \text{opposite}(p) \neq \Omega \implies \neg\text{composite}(\text{opposite}(p)))$

Metamodel Constraint 4 *No infinite chain of compositions*: $(\forall c_1, \ldots, c_n, c_{n+1} \in C \cdot (\forall i \cdot 1 \leq i \leq n \implies (\exists p \in \text{effectiveProperties}(c_i) \cdot \text{composite}(p) \wedge \text{owner}(p) = c_i \wedge c_{i+1} = \text{type}(p) \wedge \text{lower}(p) \geq 1)) \implies c_1 \neq c_{n+1}), \quad \forall n \geq 1$

Metamodel Constraint 5 *Generalization is acyclic*: $\neg(\exists e \in C \cdot (e, e) \in \{(c, d) \cdot d \in \text{generalizations}(c)\}^{+})$

Metamodel Constraint 6 *Upper multiplicity in subset properties*: $(\forall p \in P \cdot (\forall q \in \text{supersets}(p) \cdot \text{upper}(p) \leq \text{upper}(q)))$

Metamodel Constraint 7 *Subset only from owner or its superclasses* $(\forall p, q \in P \cdot p \subseteq_p q \implies \text{owner}(p) \subseteq_c \text{owner}(q))$.

Metamodel Constraint 8 *The property superset relation is acyclic*: $\neg(\exists e \in P \cdot (e, e) \in \{(p, q) \cdot q \in \text{supersets}(p)\}^{+})$

Metamodel Constraint 9 *The opposite of a subset property must be a subset*: $(\forall p, q \in P \cdot p \subseteq_p q \wedge \text{opposite}(p) \neq \Omega \implies \text{opposite}(p) \subseteq_p \text{opposite}(q))$

Metamodel Constraint 10 *No circular transitive composition with subsets*: $(\forall p \in P \cdot \text{composite}(p) \implies \neg(\exists q \in P \cdot \text{opposite}(q) \neq \Omega \wedge p \subset_p q \wedge \text{composite}(\text{opposite}(q))))$

The models

The models are defined by:

$M = (E, \text{class}, S, \text{property}, \text{slotOwner}, \text{contents})$

The model constraints

Model Constraint 1 *Valid slots in element* (1): $(\forall e \in E \cdot (\forall s \in \text{slots}(e) \cdot (\text{property}(s)) \in \text{effectiveProperties}(\text{class}(e))))$

Model Constraint 2 *Valid slots in element* (2): $(\forall e \in E \cdot (\forall p \in \text{effectiveProperties}(\text{class}(e)) \cdot (\exists! s \in \text{slots}(e) \cdot \text{property}(s) = p)))$

Model Constraint 3 *Class of elements in a slot*: $(\forall s \in S \cdot (\forall e \in \text{contents}(s) \cdot \text{class}(e) \in \text{effectiveType}(\text{property}(s))))$

Model Constraint 4 *Valid amount of elements in a slot*: $(\forall s \in S \cdot \text{lower}(\text{property}(s)) \leq \#s \leq \text{upper}(\text{property}(s)))$

Model Constraint 5 *Bidirectionality of slots*: $(\forall s \in S \cdot \text{opposite}(\text{property}(s)) \neq \Omega \implies (\forall e' \in \text{contents}(s) \cdot (\exists! s' \in S \cdot \text{slotOwner}(s') = e' \wedge \text{opposite}(\text{property}(s')) = \text{property}(s) \wedge \text{slotOwner}(s) \in \text{contents}(s'))))$

Model Constraint 6 *Overridden by model constraint* 11.

Model Constraint 7 *Composition is acyclic*: $(\forall e_1, \ldots, e_n, e_{n+1} \in E \cdot (\forall i \cdot 1 \leq i \leq n \implies e_i \in \text{parent}(e_{i+1})) \implies e_1 \neq e_{n+1}), \quad \forall n \geq 1$

Model Constraint 8 *Unordered slots*: $(\forall r, s \in S \cdot r \subseteq_s s \wedge \neg\text{ordered}(\text{property}(s)) \implies \text{contents}(r) \subseteq \text{contents}(s))$

Model Constraint 9 *Ordered slots*: $(\forall x, y \in E, r, s \in S \cdot x \in \text{contents}(r) \wedge y \in \text{contents}(r) \wedge x \preceq_r y \wedge r \subseteq_s s \wedge \text{ordered}(\text{property}(s)) \implies x \in \text{contents}(s) \wedge y \in \text{contents}(s) \wedge x \preceq_s y)$

Model Constraint 10 *Elements in a strict union*: $(\forall s \in S \cdot \text{strictUnion}(\text{property}(s)) \implies \text{contents}(s) = \bigcup\{\text{contents}(r) \cdot r \prec_s s\})$

Model Constraint 11 (*Subset*) *Only in one composite slot*: $(\forall e \in E \cdot \neg(\exists s_1, s_2 \cdot \text{slotOwner}(s_1) = \text{slotOwner}(s_2) \wedge (\text{property}(s_1) \,||_p\, \text{property}(s_2)) \wedge \text{composite}(\text{property}(s_1)) \wedge \text{composite}(\text{property}(s_2)) \wedge e \in \text{contents}(s_1) \wedge e \in \text{contents}(s_2))$

Appendix B: Mathematical notation

This appendix summarizes the notation used in this article, with some small examples.

We use naive set theory throughout the article. Sets of primitive data values are denoted with calligraphic letters: $\mathbb{B}$ is the set of boolean values, $\mathbb{Z}^{0+}$ is the set of integers $0, 1, 2, \ldots$ and $\mathbb{Z}^{+}$ is set of the set of integers $1, 2, 3, \ldots$

The expression $\#S$ is the amount of elements in a finite set S.

We use the notation $A \subset B$ to denote that A is a strict subset of B. We use $A \subseteq B$ to denote the possibility that A is a subset of or equal to B.

The expression $\mathcal{P}(A)$ is the powerset of a set A, i.e., the set of all possible subsets. For example, the powerset of $\{1, 2, 3\}$ is $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

Set comprehensions are denoted with $\{g(x) \cdot f(x)\}$, which returns a set of values $g(x)$ where $f(x)$ is true (for all possible legal values of x). The notation $\bigcup\{g(x) \cdot f(x)\}$ denotes the set consisting of all elements in all sets $g(x)$ where $f(x)$ is true.

A function $f : A \to B$ maps an element of the set A to an element of the set B. A partial function $f : A \nrightarrow B$ is not necessarily defined for all elements of A.

A binary relation R is a set of pairs (a, b). A reflexive relation is such that $(\forall a \cdot (a, a) \in R)$. A transitive relation is defined by $(a, b) \in R \land (b, c) \in R \implies (a, c) \in R$.

If we have a set of pairs R whose values are of the same domain A, we can create the transitive closure R^{+} by taking the smallest transitive relation over the domain of the values that still contains R. The reflexive closure of R is $R^{=} = R \cup \{(a, a) \cdot a \in A\}$. The reflexive transitive closure of R is $R^{*} = R^{+=}$.

A partial order $(A, \subseteq_A)$ is a set A and a binary operator $\subseteq_A$. The operator determines the partial ordering of elements; given $a \in A$ and $b \in A$, the operation $a \subseteq_A b$ is true if a occurs before b in the ordering, otherwise false. The expression $a \,||_A\, b = \neg(a \subseteq_A b) \land \neg(b \subseteq_A a)$ means that a and b are independent and cannot be compared in the partial order.

We denote by $a \prec_A b$ the fact that a is a "directly below" b in a partial order, i.e., $a \prec_A b = a \subset_A b \land \neg(\exists c \cdot c \neq a \land c \neq b \land a \subset_A c \subset_A b)$.

The partial order should not be confused with the subset operator $\subseteq$. The former is an arbitrary function that determines the partial order, whereas the latter has only one definition in set theory.

An array $(A, \prec)$ is an ordered set of elements A. It is essentially like a set of elements, except that all elements have a unique position in the array. We denote $a \prec_x b$ if element a precedes element b in a specific array x. We denote $a \preceq_x b$ if a precedes b or if $a = b$. Note that two elements can be in different orders in different arrays.

References

1. Alanen, M., Lundkvist, T., Porres, I.: Comparison of modeling frameworks for software engineering. Nord. J. Comput. **12**(4), 321–342 (2005)
2. Alanen, M., Porres, I.: Coral: a metamodel kernel for transformation engines. In: Akerhurst, D.H. (ed.) Proceedings of the Second European Workshop on Model Driven Architecture (MDA), number 17, pp. 165–170. University of Kent (2004)
3. Alanen, M., Porres, I.: Model Interchange Using OMG Standards. In: Werner, B. (ed) Proceedings of the 31st Euromicro Conference on Software Engineering and Advanced Applications, pp. 450–458. IEEE Computer Society, Aug 2005. ISBN 0-7695-2431-1
4. Alanen, M., Porres, I.: A metamodeling language supporting subset and union properties. In: Prinz, A., Tveit, M.S. (eds.) 4th Nordic Workshop on the Unified Modeling Language NWUML'2006, Jun 2006
5. Alanen, M., Porres, I.: Basic operations over models containing subset and union properties. In: Oscar Nierstrasz, D.H., Whittle, J., Reggio, G. (eds) Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), vol. 4199 of Lecture Notes in Computer Science, pp. 469–483. Springer, Berlin, Oct 2006
6. Albano, A., Ghelli, G., Orsini, R.: A relationship mechanism for a strongly typed object-oriented database programming language. In: Proceedings of the 17th Conference on Very Large Databases. Morgan Kaufman Publishers Inc. (1991)
7. Álvarez, J., Evans, A., Sammut, P: MML and the metamodel architecture. In: Whittle, J. (ed) WTUML: Workshop on Transformation in UML 2001, April 2001
8. Amelunxen, C., Rötschke, T., Schürr, A.: Graph Transformations with MOF 2.0. In: Holger Giese, Albert Zündorf (eds.) Fujaba Days 2005, September 2005
9. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Trans. Model. Comput. Simul. **12**(4), 290–321 (2002)
10. Baar, T.: Metamodels without metacircularities. L'Objet **9**(4), 95–114 (2003)
11. Back, R.-J., Grundy, J., von Wright, J.: Structured calculational proof. Technical Report 65, Turku Center for Computer Science, November 1996
12. Barbier, F., Henderson-Sellers, B., Le Parc, A., Bruel, J.-M.: Formalization of the Whole-Part Relationship in the Unified Modeling Language. IEEE Trans. Softw. Eng. **29**(5), 459–470 (2003)
13. Baresi, L., Heckel, R.: Tutorial introduction to graph transformation: a software engineering perspective. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Proceedings of Graph Transformation—First International Conf., ICGT 2002, Barcelona, Spain, vol. 2505 of LNCS. Springer, Heidelberg (2002)
14. Bézivin, J., Breton, E., Dupé, G., Valduriez, P.: The ATL Transformation-based Model Management Framework. Technical Report 03.08, University of Nantes, France (2003)
15. Bierman, G., Wren, A.: First-class relationships in an object-oriented language. In: Workshop on Foundations of Object-Oriented Languages (FOOL 2005), January 2005
16. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. Addison Wesley Professional, August 2003
17. Castagna, G.: Covariance and contravariance: conflict without a cause. ACM Trans. Program. Lang. Syst. **17**(3), 431–447 (1995)
18. Clark, T., Evans, A., Kent, S.: The metamodelling language calculus: foundation semantics for UML. In: Proceedings of the Fundamental Aspects of Software Engineering (FASE), pp. 17–31 (2001)

19. Akehurst, D.H., Kent, S., Patrascoiu, O.: A relational approach to defining and implementing transformations between metamodels. Softw. Syst. Model. **2**(4), 215–239 (2003)
20. EMF development team. The Eclipse Modeling Framework website. http://www.eclipse.org/emf
21. France, R., Rumpe, B.: Domain specific modeling, Editorial. Springer Int. J. Softw. Syst. Model. **4**(1) (2005)
22. Génova, G., del Castillo, C.R., Lloréns, J.: Mapping UML Associations into Java Code. J. Object Technol. **2**(5), 135–162 (2003)
23. Gonzalez-Perez, C., Henderson-Sellers, B.: a powertype-based metamodelling framework. Softw. Syst. Model. 5:72–90 (2006). doi:10.1007/s10270-005-0099-9
24. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
25. Henderson-Sellers, B., Barbier, F.: Black and white diamonds. In: France, R., Rumpe B. (eds) UML'99—The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28–30. 1999, Proceedings, vol. 1723 of *LNCS*, pp. 550–565. Springer, Heidelberg (1999)
26. Jiang, J., Systä, T.: Exploring differences in exchange formats—tool support and case studies. In: Seventh European Conference on Software Manteinance and Reengineering. IEEE Computer Society, March 2003
27. Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Bologna, Italy (2006)
28. Kalnins, A., Barzdins, J., Celms, E.: Basics of model transformation language MOLA. In: Workshop on Model Transformation and Execution in the Context of MDA (ECOOP 2004), June 2004
29. Kleppe, A.: April 2003. Discussion on the mailing-list pumllist@cs.york.ac.uk
30. Liskov, B.: Keynote address—data abstraction and hierarchy. SIGPLAN Not **23**(5), 17–34 (1988)
31. Lundell, B., Lings, B., Persson, A., Mattsson, A.: UML model interchange in heterogeneous tool environments: an analysis of adoptions of XMI 2. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), volume 4199 of Lecture Notes in Computer Science. Springer, Berlin (2006)
32. Netbeans. Netbeans Metadata Repository (NMR). Available at http://mdr.netbeans.org/
33. Nickel, U.A., Niere, J., Zündorf, A.: Tool demonstration: the FUJABA environment. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE), pp. 742–745. ACM Press (2000)
34. Nytun, J.P., Prinz, A., Kunert, A.: Representation of levels and instantiation in a metamodelling environment. In: Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language NWUML'2004, pp. 1–17 (2003)
35. OMG. Meta Object Facility, version 1.4, April 2002. Document formal/2002-04-03. Available at http://www.omg.org/
36. OMG. XML Metadata Interchange (XMI) Specification, version 1.2, January 2002. Available at http://www.omg.org/
37. OMG. XML Metadata Interchange (XMI) Specification, version 2.0, May 2003. Available at http://www.omg.org/
38. OMG. MOF 2.0 Query/View/Transformation Final Adopted Specification, November 2005. OMG Document ptc/05-11-01. Available at http://www.omg.org/
39. OMG. UML 2.0 Superstructure Specification, August 2005. Document formal/05-07-04. Available at http://www.omg.org/
40. OMG. XML Metadata Interchange (XMI) Specification, version 2.1, September 2005. Available at http://www.omg.org/
41. OMG. Meta Object Facility (MOF) Core Specification, version 2.0, January 2006. Document formal/06-01-01. Available at http://www.omg.org/
42. OMG. UML 2.0 Infrastructure Specification, March 2006. Document formal/05-07-05. Available at http://www.omg.org/
43. OMG Architecture Board. Model Driven Architecture—A Technical Perspective, 2001. OMG Document ormsc/01-07-01. Available at http://www.omg.org/
44. Octavian Patrascoiu. YATL:Yet Another Transformation Language. In: Proceedings of the 1st European MDA Workshop, MDA-IA, pp. 83–90. University of Twente, The Nederlands (2004)
45. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations, vol. 1. Foundations. World Scientific (1997)
46. Scheidgen, M.: On Implementing MOF 2.0—New Features for Modelling Language Abstractions. July 2005. Available at http://www.informatik.hu-berlin.de/ scheidge/
47. Steel, J., Jézéquel, J.-M.: Typing Relationships in MDA. In: Akehurst, D.H. (ed) Proceedings of the Second European Workshop on Model Driven Architecture (EWMDA), number 17, Canterbury, Kent CT2 7NF, UK, Sep 2004. University of Kent
48. Steel, J., Jézéquel, J.-M.: Model typing for improving reuse in model-driven engineering. In: MoDELS, pp. 84–96 (2005)
49. Sutton, A.: Open Modeling Framework. Available at http://www.sdml.info/projects/omf/
50. ATLAS Team. Atlantic Metamodel Zoo (2006). http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/
51. Tratt, L.: The MT model transformation language. In: Proceedings of ACM Symposium on Applied Computing, pp. 1296–1303, April 2006
52. Varró, D.: Automatic program generation for and by model transformation systems. In: Kreowski, H.-J., Knirsch, P. (eds) Proceedings of AGT 2002: Workshop on Applied Graph Transformation, pp. 161–173, Grenoble, France, April 12–13 (2002)
53. Varró, D., Pataricza, A.: VPM: a visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. J. Softw. Syst. Model. **2**(3), 187–210 (2003)
54. Varró, D., Varró, G., Pataricza, A.: Designing the automatic transformation of visual languages. Sci. Comput. Program. **44**(2), 205–227 (2002)
55. Winter, A., Kullbach, B., Riediger, V.: An overview of the GXL graph exchange language. In: Revised Lectures on Software Visualization, International Seminar, pp. 324–336. Springer, London (2002)

Author's Biography

Marcus Alanen received his M.Sc. degree in Computer Engineering in 2002 at Åbo Akademi University, Finland. He is currently working on his Ph.D. at the same university, researching metamodeling concepts, diagram interchange, difference calculation and serialization of models.

Ivan Porres received his M.Sc. degree in Computer Science in 1997 at the Polytechnic University of Valencia, Spain and in 2001 his Ph.D. in Computer Engineering at Åbo Akademi University in Turku, Finland. His thesis, "Modeling and Analyzing Behavior in UML", shows how to use different formal methods to ensure the correctness of software modeled using the UML. Currently, he is a docent in Software Engineering and works as acting professor at the Department of Information Technologies at Åbo Akademi University where he researches the topics of model-driven software development and software process improvement.