Appeared in the Journal on Software and Systems Modeling, Volume 5, Number 4, pp. 369-385, December 2006

Matters of (Meta-) Modeling

Thomas Kühne

Darmstadt University of Technology, Darmstadt, Germany
e-mail: `kuehne@informatik.tu-darmstadt.de`

Abstract With the recent trend to model driven engineering a common understanding of basic notions such as "model" and "metamodel" becomes a pivotal issue. Even though these notions have been in widespread use for quite a while, there is still little consensus about when exactly it is appropriate to use them. The aim of this article is to start establishing a consensus about generally acceptable terminology. Its main contributions are the distinction between two fundamentally different kinds of model roles, i.e. "token model" versus "type model"[1], a formal notion of "metaness", and the consideration of "generalization" as yet another basic relationship between models. In particular, the recognition of the fundamental difference between the above mentioned two kinds of model roles is crucial in order to enable communication among the model driven engineering community that is free of both unnoticed misunderstandings and unnecessary disagreement.

Key words model driven engineering, modeling, metamodeling, token model, type model

1 Introduction

Everytime a new research area gains momentum, the task of defining its central notions needs to be addressed. Communities can take a surprisingly long time to come to an agreement about what notions like "object" and "component" should encompass. Although such efforts can be tedious and are renown for causing research meetings to stall on the "definition problem", they are necessary in order to enable unambiguous communication among community members. A number of efforts to establish an unambiguous vocabulary (e.g., [1], [2]) testify to the need for a shared conceptualization in model engineering. If the community continues to maintain different ontologies for the basic terms of their discipline, any communication may create the illusion of agreement where there is none, i.e., unnoticed misunderstandings, and raise barriers of communication where they are just accidental.

In the following we will focus on the term "model" in the context of model driven engineering. Our models are thus all language-based in nature, unlike, e.g., physical scale models and they describe something as opposed to models in mathematics which are understood as interpretations of a theory [3].

In an attempt to define the scope of the notion "model" we should consider how it has been traditionally used in software engineering. From this perspective, a model is an artifact formulated in a modeling language, such as UML [4], describing a system through the help of various diagram types. Typically, such model descriptions are graph-based and are rendered visually.

In our model driven engineering context such a characterization would be too narrow. Other artifacts, such as Java programs, are considered to be models as well, since they can also be understood as describing systems (e.g., all possible execution traces of a program). This liberal use of "model" is the result of applying the powerful principle of unification ("Everything is a model" [5]). Its intent is to get as much out of the new development paradigm as possible. For instance, if transformations are considered models as well [5] then—since the fundamental operation in model driven engineering is a "model to model" transformation—using the standard general infrastructure, one may obtain a particular transformation by transforming another one.

Obviously, even if "everything is a model" were unconditionally true, this would not relieve us from the task of defining the fundamental model relationships. At least in a relative way one needs to be able to speak about different roles (such as system, model and element) and corresponding relationships (such as representation and instantiation), independently of what is and what is not considered a model, e.g., whether or not the modeled system is a model itself, or the elements of a model are models again, etc.

Not before the same "notion definition" task was completed for the basic notions of object-orientation, i.e., not before

[1] The terms "type" and "token" have been introduced by C.S. Peirce, 1839–1914.

notions like instantiation and inheritance were fully understood, the full potential of object-orientation was unfolded. For instance, a solid notion of subtyping as a discipline for inheritance is crucial to build safely reusable software. We need to achieve the same clarity and consensus for the basic notions of model driven engineering as well, in order to unlock its full potential.

The remainder of this article first attempts to home in on a characterization of "model" in the context of model driven engineering that everyone may subscribe to. Next we will distinguish two fundamentally different kinds of model roles. Only after the difference between these two kinds has been made explicit, will we be able to further define basic model notions and properties, such as "metamodel" and "generalization" between models. We complement this discussion by relating the notions "metamodel" and "language" with each other and then conclude.

2 What is a Model?

In this section we attempt to define a scope for the notion "model" that is broad enough to include everything useful, but narrow enough so that it does not become useless. After all, if a notion includes everything, it loses its discrimination property. As an analogy, practically everything could be characterized as an "object" but as a technical term it only provides value in communication if not everything is included by the term "object". Likewise the notion "model" should include "transformation" but not in an opportunistic manner (if it is considered useful then it is deemed correct) but in a way that includes "transformation" in a well defined scope of sufficient size but without unnecessary breadth. In our context the following definition is useful:

> A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made.

While the aim of this article is not to present any proofs or a complete formalization of modeling, we nevertheless try to disambiguate and concisely present the essence of informal textual statements with some formal syntax and hence denote the relationship between a system $\mathcal{S}$ and a model of it as

$$\mathcal{S} \lhd \mathcal{M}. \tag{1}$$

In general, $\lhd$ establishes a many-to-many relationship since one model may describe several systems and one system may be described by several models. Mathematically, $\lhd$ therefore is a binary relation. A subset of this relation is the representation[2] relation ρ, hence

$$\rho(\mathcal{S}, \mathcal{M}) \rightarrow \mathcal{S} \lhd \mathcal{M}.$$

While the "model-of" ($\lhd$) relation includes any accidental, legally conforming "system / model" pair, relation ρ is meant to capture only such pairs where the model is specifically intended to represent the corresponding system. We are thus able to state that a model $\mathcal{M}_2$ models another model $\mathcal{M}_1$, but that both represent a single original system:

$$\mathcal{S} \lhd \mathcal{M}_1 \wedge \rho(\mathcal{S}, \mathcal{M}_1)$$
$$\mathcal{M}_1 \lhd \mathcal{M}_2 \wedge \rho(\mathcal{S}, \mathcal{M}_2)$$
$$\text{as opposed to } \rho(\mathcal{M}_1, \mathcal{M}_2).$$

Fig. 1 shows a corresponding example featuring two map models. Fig. 1 and most other figures use standard UML notation [4] with the usual meaning of objects, classes, associations, dashed dependency lines with "instance-of" stereotypes to denote instantiation, etc. Only a few non-UML elements are used for illustrative purposes. Associations are often annotated with the notation we successively introduce in order to connect the textual definitions to the examples shown in figures. For easy reference, Tab. 2 serves as a final summary to the notation introduced and Fig. 7 illustrates the most important relationships accordingly.

In order to be able to discuss various model properties later on, we also assume an abstraction function α that produces a model from a system:

$$\mathcal{M} = \alpha(\mathcal{S}). \tag{2}$$

In the spirit of [6], we are assuming that $\mathcal{S}$ is already available in a formal representation. For instance, a structured set "(S, r_s)" with elements (from S) and relationships (from r_s) between those elements is an adequate choice.

If $System$ is the real (e.g., physical) system under study, then $\mathcal{S}$ can be thought of as being generated by a process "modeling" from $System$. While one could argue that process "modeling" embodies the very operation we are trying to study, our approach works without any loss of generality. For our purposes it is irrelevant whether the system we abstract from, conceptualize, etc., is real or already is a representation of a real system.[3] Any philosophical and epistemological issues of the process "modeling" , e.g., how to extract structure and properties from a real system with practically infinitely many properties and possibly unobservable behavior into a formal representation, are not of interest to us in this context, since we are focusing on "model" as a technical term in model driven engineering rather than investigating the process of creating representations of reality.

2.1 Model Features

According to Stachowiak [7] a model needs to possess three features (see Tab. 1). The first two features are covered at once if one informally speaks of a model as a "projection", as this implies both that something is projected (the original)

[2] Here, we use "representation" as in "be a placeholder / representative for", not to be confused with "presentation" as in "concrete syntax for communication purposes".

[3] Actually, we do lose the property of real systems to be non-language-based, i.e., not being an expression of a language, but this does not become relevant before section 5.

mapping feature	A model is based on an original.[4]
reduction feature	A model only reflects a (relevant) selection of an original's properties.
pragmatic feature	A model needs to be usable in place of an original with respect to some purpose.

Table 1 Model Features according to Stachowiak.

and that some information is lost during the projection. Formally, relating to equation 2:

$$\alpha = \tau \circ \alpha' \circ \pi. \tag{3}$$

Model abstraction (α) then consists of projection (π), some further abstraction (α') on elements (including relationships), and a translation τ to another representation, i.e., the modeling language. With projection π we associate any filtering of elements both reducing their number and individual information content. Projection π is an injective homomorphism, i.e., a structure preserving operation.

Following [8], we regard a function $h : X \mapsto Y$ as a homomorphism if

$$h(a \oplus b) = h(a) \otimes h(b)$$

where $\oplus$ is an operation on X and $\otimes$ is an operation on Y. As a result, π preserves the structure of the original for those parts that it retains and creates a one-to-one relationship between target and (a subset of the) source.

For later reference, we label the reduced intermediate result between projection and further abstraction as:

$$\mathcal{S}_r = \pi(\mathcal{S}) \tag{4}$$

Exactly what information of the system is left after projection, depends on the ultimate purpose of the model—Stachowiak's third feature—the pragmatic useability of the model, i.e., who the model is for and for what purpose. Steinmüller [9] even includes both the sender and the recipient as being relevant. According to Steinmüller a model is information

- on something (content, meaning),
- created by someone (sender),
- for somebody (receiver),
- for some purpose (usage context).

A common purpose for a model is that it is used in place of the system. Any answers obtained from the model should then be the same as those given by the system provided the model is adequate [1] / correct [10]. Typically, the motivation for using a model is cost-saving as it is often cheaper and/or quicker to obtain answers from a model than from the system. Often models are even known to be imprecise or false in some respect, but this does not automatically mean that they are inadequate. Such imprecisions, at best, may not affect the properties of interest at all or, slightly worse, may just make their evaluation more uneconomic or, worse, skew them but to an acceptable extent only.

[4] We have so far referred to the original as the "system".

2.2 Motivation for Modeling

In software engineering, models typically come in two flavors: *descriptive* and *prescriptive*. Descriptive models are used to capture some knowledge, e.g., requirements, a domain analysis, etc.

Prescriptive models (aka, "specification models" [10]) are used as blueprints (construction plans) for system designs, implementations, etc. Nowadays, the main purpose of blueprints is to support planning and early validation, i.e., finding errors as early as possible and partially evaluating a system before it is realized, but the value obtainable from their prescriptive nature is limited because of the lack of rigor and preciseness regarding their meaning. It is one of the primary goals of model driven engineering to shift the emphasis from informal, non-binding models to rigorous, binding models.

Note that the idea of a model as a construction plan is, in principle, in conflict with the required "mapping" feature. There is no "original" to map the model to (yet). Still, Webster's new encyclopedic dictionary [11] includes the following definition of "model":

a) a theoretical projection of a possible or imaginary system.

In other words, the "original" might be something yet to be built or it may remain completely imaginary. Only the former possibility seems to be of relevance in our context and we still accept construction plans as models since there clearly is an intended system the model will represent. In order to deemphasize the connotation of "original" as preceding the model in time with respect to existence, a sometimes more suitable term for "original" is "subject".

2.3 Are Transformations Models?

So far, our notion of "model" is in accordance with established definitions, but does it include unconventional interpretations of model, such as "transformation"? In fact, a (model-to-model) transformation is indeed *information on* a mapping from one model to another, *created* by a transformation engineer, *for* the transformation engine, *in order to* automate a translation process. So a transformation refers to an *original* (the actual mapping links between actual models) and only reflects *relevant properties* of the original since it does not spell out all individual mapping links but only describes the mapping scheme in terms of the model languages. This is true as long as one understands "transformation" as "description of a transformation function". If we interpret "transformation" to be all the actual links from all elements from the source model to all elements of the target model then transformations should not be considered as models, as we can then no longer point out any reduction feature.[5] This view is in accordance with the terminology offered by [2], where actual

[5] Unless, of course, the transformation is a model of a real transformation going on between the originals of the respective source and target models. However, this is an exceptional case we are not further considering here.

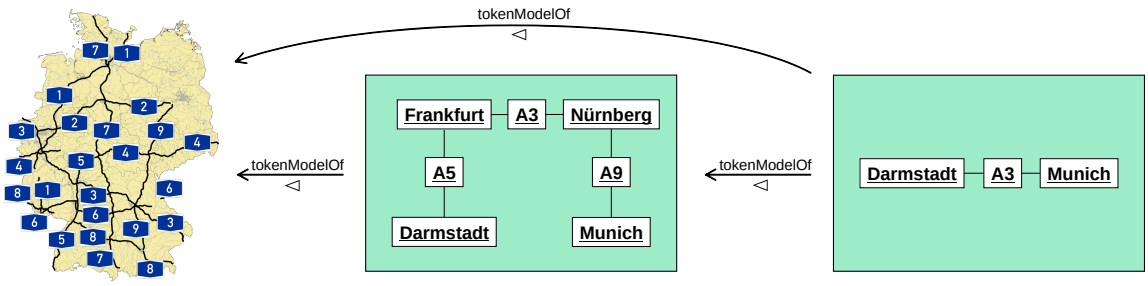

Fig. 1 Token Models and Model Transitivity

connections/links are referred to as "transformationInstance" and only descriptions of "transformationFunctions" are referred to as "transformationModels".

2.4 A Copy is not a Model

Obviously, for the sake of making as many artifacts as possible eligible to be considered models, we could drop the demand on models to have a reduction feature. However, this could be the threshold beyond which the notion starts deteriorating into something more or less meaningless.

Note that we are only able to speak about the absence of reduction because our subjects are finite representations already. Any representation of a real world subject automatically implies reduction and thus can be granted model status.

In the context of descriptions, whose subjects are finite formal representations, we may even consider accepting another of the definitions for "model" from Webster's new encyclopedic dictionary [11]—

b) a small but exact copy of something

—as long as "exact" refers to the properties one wants to retain but is not understood to mean "complete".

If I build a car according to an original being precise in every minute detail, I have not constructed a model but a copy.[6] If I use the copy in a crash test, I have not performed a model simulation, but a real test run. Exact copies neither offer the advantages of models (typically cost reduction) nor do they entail their disadvantages (typically inaccuracy with regard to the original). In other words, "no abstraction" → "no model". If, in order to maximize the unification principle, we would still accept copies as models then we should at least refer to them as *degenerate models*.

3 Kinds of Model Roles

Intriguingly the discussion so far has not had to take into account the existence of two fundamentally different kinds of models. If in personal communication one expert thought of the one kind and another expert thought of the other kind, so far they would have always been in agreement. However, as soon as further characterizations are attempted, such as "transitivity of the 'model-of' relationship" or "under which circumstances is a model a metamodel?", the experts would start disagreeing and may only consolidate their views again when discovering their different mindsets.

There are of course many ways in which one can distinguish models, such as "product versus process models" or "static versus dynamic models" but for the following discussion these differences are irrelevant. The two kinds of models which are able to create communication chasms between experts talking about basic modeling notions are token and type models. As the section title indicates these kinds are not absolute properties of models but depend on their relationship to the system.

3.1 Token Models

A typical example for a token model is a map (see the middle part of Fig. 1). Note that here (on the left hand side of Fig. 1) and elsewhere we use depictions of real world systems for model subjects for illustrative purposes only, so that the latter can be better recognized as subjects as opposed to being regarded as models themselves. In our formal treatment, however, we continue to assume model subjects to be representations already.

Elements of a token model capture singular (as opposed to universal) aspects of the original's elements, i.e., they model individual properties of the elements in the system.

When using UML, a natural choice for creating a token model is the object diagram since it captures the system's elements that one is interested in in a one-to-one mapping and *represents* them with their individual attribute values. Note, however, that depending on the nature of the subject, class diagrams may also be appropriate (see section 3.2, in particular Fig. 3).

Formally, with respect to equation 3 we have

$$\alpha' = id \tag{5}$$

In other words, the abstraction process for creating token models involves no further abstraction beyond projection and translation (see equation 3).[7] As a consequence, elements of

[6] In our context we can also refute the model status of the copy by observing that it is not a language-based description of an original.

[7] Here, we restrict our notion of "model" to pure abstractions of their subjects. Any additional information they might contain—

a token model are designators for those elements in the system $\mathcal{S}$ which are also retained in the reduced system $\mathcal{S}_r$ (see equation 4). We therefore have a one-to-one correspondence between relationships and elements in the model $\mathcal{M}$ and a subset of these in system $\mathcal{S}$. This property implies that the "token-model-of" relationship must be transitive. A chain of token models can be regarded as a chain of designators, linearly referencing each other and then, ultimately, the system. Hence, the designators of the last token model transitively reach down to a subset of the system's elements.

It is instructive to realize that coarsely capturing elements of a system in a model (through π)—e.g., not distinguishing between two-lane or three-lane motorways and representing them all as just plain "motorway" elements—must not be confused with generalization. The former is a projection of elements onto the same number of elements designating the originals in an abstract way, i.e.,

$$\pi(e_1) = \pi(e_2) \nrightarrow e_1 = e_2$$

whereas generalization is the union of two or more special concepts into one general concept. However, there is of course a correspondence between the equivalence relationship $\sim_\pi$ mapping different source elements onto the same target element–

$$e_1 \sim_\pi e_2 \rightarrow \pi(e_1) = \pi(e_2)$$

–and generalization: The extension of a general concept may exactly define the elements which are considered to be equivalent to each other by $\sim_\pi$, in the presence of more differentiating, special concepts.

In the context of this article, a "concept" implies an intensional abstraction of predicates which characterize the same elements in a description independent manner. For instance, we know that $UnfeatheredBipeds(X) \sim ScriptUsingMammal(X)$ so that one may refer to the characterized elements with the concept *HumanBeing*(X). If C is a concept we use $\varepsilon(C)$ to refer to its extension (all elements falling under the concept C) and $\iota(C)$ to refer to its intension (a conjunction of predicates characterizing whether an element belongs to the concept or not) [12], so that

$$\varepsilon(C) = \{x \mid P(x)\}, \text{ where } P = \iota(C)$$

Hence, with respect to "generalization" we can state intensionally—

$$\forall x : \iota(C_{special})(x) \rightarrow \iota(C_{general_1})(x)$$

—and extensionally:

$$\varepsilon(C_{special}) \subseteq \varepsilon(C_{general}).$$

In UML parlance token models are sometimes referred to as "snapshot models" since they can be used to capture a single configuration of a dynamic system. Their fine-grained representation of a system—retaining system elements in a one-to-one fashion—makes them ideal for capturing detail that changes dynamically in time. Other possible names for token models are "representation model" (due to the direct representation character), "instance model" (since the model elements are instances as opposed to types), "singular model" (because the elements designate individuals rather than universals), or "extensional model" (as they are enumerative with respect to system elements).

In software engineering, stereotypical usages for token models include the capturing of initial system configurations, or system snapshots as a basis for simulations (e.g., regarding performance). Token models are also often what people have in mind when talking about models in general. The often used example of a building plan for a house, is a token model.

potentially completely unrelated to the subject—are not considered by this treatment.

3.2 Type Models

As we have seen in the previous section, token models have many useful applications. However, they do little to condense complex systems to concise descriptions, due to their one-to-one representation of elements in the (relevant part of the) system. Type models are much more economic in this respect. For good reasons the human mind exploits the power of type models by using object properties (e.g., four legged, furry, sharp teeth, and stereovision) to *classify* objects (e.g., as a predator) and then draw conclusions according to properties known about the object class (e.g., "dangerous"). This way the human mind does not need to memorize all particular observations and arrive at decisions afresh, but just collects concepts and their universal properties [13].

Most models used in model driven engineering are type models. In contrast to token models, type models capture the *universal* aspects of a system's elements by means of *classification*.

Fig. 2 shows (at the top) a type model for the modeled country, using UML's natural diagram kind for type models: the class diagram. Instead of representing all the particular elements and their links, the type model captures the types of interest and their associations only. Thus "schema model", "classification model", "universal model", or "intensional model" are further appropriate names.

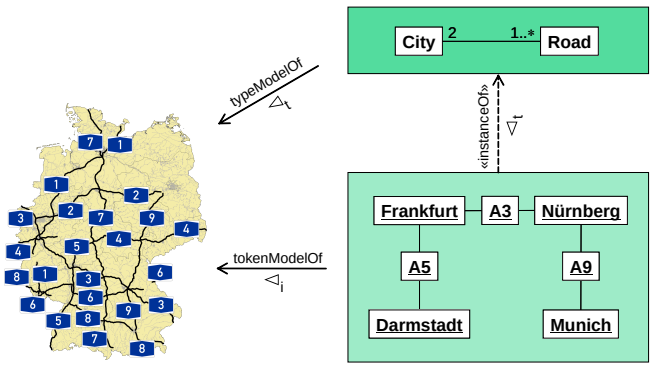

Fig. 2 Kinds of Model Roles

Formally, with respect to equation 3 we have—

$$\alpha' = \Lambda \tag{6}$$

—where Λ is a classification function, classifying elements (including relationships), which are considered equivalent to each other with respect to certain properties, under one respective type. Hence, the complete abstraction function for creating type models involves classification in addition to projection and translation (see equation 3).

Function Λ is also a homomorphism. If it classifies every system element (from $\mathcal{S}_r$) into its own singleton set then it is even an isomorphism. Of course, the usefulness of type models stems from the fact that this is typically not the case.

One may check whether a model $\mathcal{M}_{token}$ conforms to another model $\mathcal{M}_{type}$ (e.g., whether the bottom-right token model in Fig. 2 conforms to the type model above it), by attempting to construct a homomorphism Λ from $\mathcal{M}_{token}$ to $\mathcal{M}_{type}$.

Such a homomorphism Λ (see equation 6), implies an equivalence relation $\sim_\Lambda$ on $\mathcal{M}_{token}$, defining which objects and relationships are to be considered equivalent, i.e., be of the same type:

$$e_1 \sim_\Lambda e_2 \leftrightarrow \Lambda(e_1) = \Lambda(e_2).$$

Hence, $\mathcal{M}_{type}$ may be regarded as the quotient of $\mathcal{M}_{token}$ with respect to the equivalence relation $\sim_\Lambda$.

$$\mathcal{M}_{type} = \mathcal{M}_{token} \ / \sim_\Lambda$$

Since type models are created by classification we may also say that model $\mathcal{M}_{token}$ is an "instance of" model $\mathcal{M}_{type}$.

Before we proceed to discuss why recognizing the difference between token and type models is important, we should clarify that being a token or a type model depends on the relationship to the modeled system, not on any intrinsic model property.

3.3 Why Roles?

Fig. 3 shows (at the middle top) a model that is usually considered a type model as its elements designate universals, i.e., classify individual objects existing during the runtime execution of a Java program. However, at the same time it is also a token model for the corresponding Java classes. The class diagram does not capture the universal aspects of the Java classes but directly represents them in a one-to-one mapping.

Hence, one needs to be careful to not judge the role of a model only by its contained elements. Consider the product model of a pet store. Whereas normally an element named "Collie" would represent a concept with an extension, i.e., many collie instances, in the case of the pet store it is simply an object representing one of the many animal types one may order. "Collie" then just designates an individual (one choice in the shop) and hence the corresponding model is a token model despite the fact that "Collie" is usually associated with a type.

Conversely, the use of an element "Lassie" typically indicates that the respective model is a token model for particular collies. Yet, it could be a type model for actual collie instances, in which "Lassie" classifies all those collies that could play the role of the movie character "Lassie". Indeed, famous "Lassie" was brought to life by many "Lassie" dog actors. Once again, the standard use of "Lassie" as an object is not a reliable indicator of the actual nature of the corresponding model.

In characterizing a model as being either a token or a type model, one must therefore always specify with respect to which model subject. More precisely, as clearly demonstrated by Fig. 3, models as such, may not be characterized at all, but one may only characterize the *reference mode* of a model with respect to a subject.

3.4 Classification versus Generalization

Equations 5 and 6 show that element abstraction (α') resolves to id for token, and to Λ for type models respectively. In both cases one might be tempted to introduce Γ as a generalization function and consider $\alpha' = \Gamma$ for token, and $\alpha' = \Gamma \circ \Lambda$ for type models respectively.

The generalization function Γ maps equivalent subject elements onto the same model element, using some equivalence relation $\sim_\Gamma$. This must not be confused with classification (Λ) since the intention of the latter is to obtain a universal for equivalent elements, whereas the intention of generalization is to increase the extension of already existing universals. In other words, even though Γ also maps many *concepts* to one (super-) *concept*, it is not the same operation as classification which maps many *elements* to one *concept*.

In the following we will attach a subscript "i" (as in "instances", aka "tokens") to the "model-of" relationship, i.e., write $\mathcal{S} \lhd_i \mathcal{M}$, in order to signal that model $\mathcal{M}$ can be regarded as a *token model* of system $\mathcal{S}$. We will use a subscript "t" (as in "types"), i.e., write $\mathcal{S} \lhd_t \mathcal{M}$, in order to signal that $\mathcal{M}$ can be regarded as a *type model* of system $\mathcal{S}$. Obviously $\lhd_i$ and $\lhd_t$ are constrained in the following way: If a model $\mathcal{M}$ can be regarded as a token model for a system $\mathcal{S}$ and the system has an instance $\mathcal{S}_i$ then $\mathcal{M}$ is a type model for $\mathcal{S}_i$, i.e.,

$$\mathcal{S} \lhd_i \mathcal{M} \wedge \mathcal{S}_i \lhd_t \mathcal{S} \rightarrow \mathcal{S}_i \lhd_t \mathcal{M}.$$

Some interesting observations can be made considering Fig. 3 with $\mathcal{S}$ referring to "Java Classes" and $\mathcal{S}_i$ referring to "Java Runtime" (see top-right labels on boxes in Fig. 3). If

$$\mathcal{S}_i \lhd_t \mathcal{S} \wedge \mathcal{S} \lhd_i \mathcal{M}, \text{ and}$$
$$\mathcal{M}_{super} = \Gamma(\mathcal{M}) \text{ then}$$

– $\mathcal{S}_i \lhd_t \mathcal{M}_{super}$: the supermodel $\mathcal{M}_{super}$ is a type model for $\mathcal{S}_i$. This is expected, as $\mathcal{S}_i$ can be viewed as a direct instance of $\mathcal{M}$ and thus also as an indirect instance of $\mathcal{M}_{super}$.

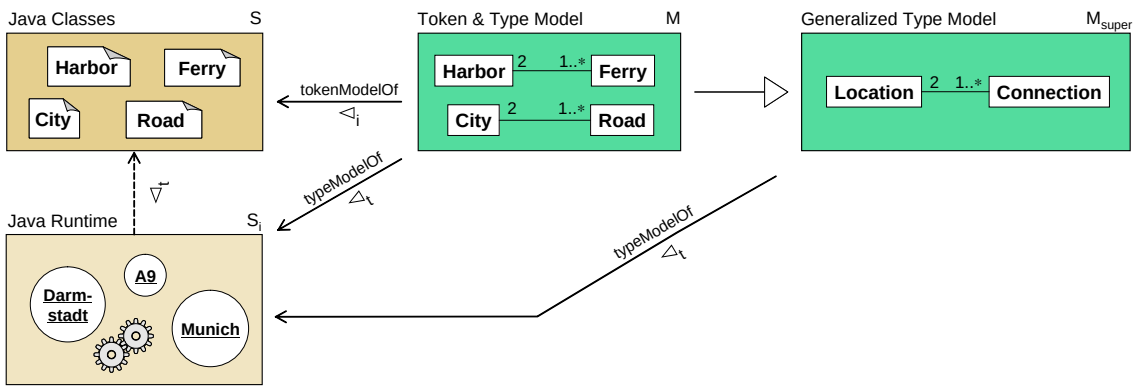

Fig. 3 Model Reference Modes and Generalization

- $\neg(\mathcal{S} \lhd_i \mathcal{M}_{super})$: the supermodel $\mathcal{M}_{super}$ is not a token model of its submodel's subject ($\mathcal{S}$), assuming that all elements in $\mathcal{S}$ should still be represented. Of course, if we were prepared to ignore some elements, e.g., "Harbor" and "Ferry" then $\mathcal{M}_{super}$ could be considered a token model of $\mathcal{S}$. However, if we still want to represent *all* elements then one element in $\mathcal{M}_{super}$ would have to represent two elements in $\mathcal{S}$, which is not possible using a token model.
- $\neg(\mathcal{M} \lhd_t \mathcal{M}_{super})$: the supermodel $\mathcal{M}_{super}$ is of course not a type for $\mathcal{M}$, as generalization is not classification.
- $\neg(\mathcal{M} \lhd_i \mathcal{M}_{super})$, the supermodel $\mathcal{M}_{super}$ is not a token model of $\mathcal{M}$, as its elements do not represent the elements of $\mathcal{M}$ in a one-to-one fashion. Note, that it is possible to reinterpret $\mathcal{M}_{super}$ to be a reduced version of $\mathcal{M}$, thus completely ignoring the way $\mathcal{M}_{super}$ was obtained. Under this assumption, i.e., that one does not intend to capture all elements of $\mathcal{M}$ but only one representative for each generalization in $\mathcal{M}_{super}$, it is indeed possible to state $\mathcal{M} \lhd_i \mathcal{M}_{super}$, bearing in mind that this is a complete reinterpretation of $\mathcal{M}_{super}$'s original role as the supermodel for $\mathcal{M}$.

If one considers system *representation* ($\rho(\mathcal{S}, \mathcal{M})$) and model *instantiation* ($\mathcal{M}_1 \lhd_t \mathcal{M}_2$) as "basic notions in model engineering" [5], then the above discussion makes it apparent that model *generalization* is another basic notion whose interplay with the first two notions needs to be defined. We have seen that to do this it was crucial to distinguish between token and type model roles.

Coming back to our original question of whether generalization might be admissible as an additional abstraction function we can now answer the question for token and type models respectively: For token models the answer is "no", i.e., $\alpha' = id$ is mandatory. Allowing generalization would make them type models of their subjects. As we have seen above, generalization only makes sense for type models.

For type models the answer is a twofold "yes": First, one may just use generalization ($\alpha' = \Gamma$) with the prerequisite that the subject must have a type model role, since if the subject has a token model role only (consider model $\mathcal{M}$ from Fig. 3 and delete "Java Runtime") then the resulting super model (here $\mathcal{M}_{super}$) does not represent a subject, neither as a type- nor as a token model.

One might of course consider establishing a new subject-model relationship, e.g, "abstract token model". In fact the resulting elements within the model would look like the elements in role-level collaborations of UML interaction diagrams. Yet, clearly such "abstract objects" or "roles" are neither objects (they represent more than one element from the subject) nor traditional types (they don't classify elements from the subject in an intensional manner as types/classes do). The best interpretations the author can think of for such entities are that they are

- placeholders for true objects, i.e., constrained (by attribute values) variables. An application for such placeholders are interaction diagrams used to represent program code.
- stereotypical objects, which result from using an alternative projection function π' that is not constrained to maintain a one-to-one mapping, but may project many elements from the system onto the same element in the model. An application for such stereotypical objects are object diagrams which are not meant to be actual system snapshots, but illustrations of how object roles perform certain interactions in general.

The second "yes" with respect to using generalization in the abstraction function for type models relates to the combination of generalization together with classification ($\alpha' = \Gamma \circ \Lambda$) to obtain a generalized version of a type model. Note that the same result can be achieved by a Λ' that directly maps to the generalized types. Conversely, we can state that any type model can be produced by first creating an isomorphic type model from the subject (through $\Lambda_{singleton}$), where each element is represented with a singleton type, and then generalizing the resulting types (through an adequate Γ).

Now that we have established the different characteristics of token and type model roles and also investigated their interplay with three basic model relationships (representation, instantiation, and generalization), we are in a position to answer the question when it is appropriate to characterize a model as a metamodel.

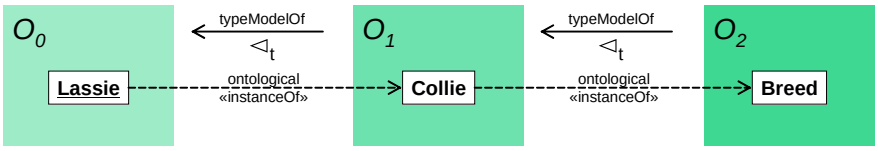

Fig. 4 Ontological Metamodeling

4 What is a Metamodel?

A literal analysis of "metamodel" suggests to investigate what the prefix "meta" signifies in other, similar contexts. Apparently the prefix "meta" is used whenever an operation is applied twice. For instance, a discussion about how to conduct a discussion is a "meta-discussion", or learning general learning strategies while learning a particular subject is "meta-learning". As a final example consider mathematicians like Hilbert who were concerned about a proper founding of mathematics and worked on subjects like proof theory in the 19th century. In order to make sure that ordinary mathematics could be performed reliably, they used mathematical methods, which is why this new subject area was coined "meta-mathematics", as mathematical methods were applied to mathematics itself.

In summary, the prefix "meta" is used before some operation f in order to denote that it was applied twice. Instead of stating "f-f", as in "class-class" one states "meta-f", e.g., "meta-class". For any further application of the operation, another "meta" prefix is added to yield "meta-meta-class", etc.

Indeed, we can find many supporting statements defining "metamodel" as implying that "modeling" has taken place twice, e.g.,

"[A metamodel is] a model of models" [14].

Also—

"A model is an instance of a metamodel" [15].

—implies that a metamodel is a model of another model.

However, if we look at the right hand side model of Fig. 1, showing a model of the model in the middle of Fig. 1 (we might be talking about a map that uses a larger scale or just provides less information than the original map), it does not seem justified to label it a "metamodel". After all, it enjoys the same relationship to the original system as its subject model, whereas real "metaness" involves some "detachment" with respect to the original. For instance, "meta-discussions" take one further away from the ordinary discussion and "meta-learning" has no immediate effect on learning a particular subject. Even though, regarding Fig. 1, we have $\mathcal{S} \lhd^2 \mathcal{M}$, i.e., we need two steps to get back to the original system, we have to refrain from accepting $\mathcal{M}$ as a metamodel because we also have the overarching link, i.e., $\mathcal{S} \lhd \mathcal{M}$, identifying $\mathcal{M}$ as an ordinary (non-meta-) model.

Indeed, when we characterize "metaness" as a two-level detachment of the original—through the double application of some operation f—we need to exclude transitive f's. Generalizing a superclass, for instance, yields another superclass only, even though we might be tempted to first construct "super-superclass" and then read that as "meta-superclass". Due to the transitivity of generalization, however, "super-super" is just the same as "super". Hence, any relation between two entities, which is going to be used to build up a meta-entity must not be transitive.

In order to define this formally, we use relation composition ($R \circ S$) and compose a relation with itself:

$$e_1 R^n e_2 = \begin{cases} \exists e : e_1 R^{n-1} e \wedge\ e R\, e_2 & , n > 1 \\ e_1 R\, e_2 & , n = 1 \end{cases}$$

Intuitively, $e_1 R^n e_2$ means that there is a path of length n from e_1 to e_2 within relation R. The standard "transitive" property for relations may hence be expressed as $e_1 R^2 e_2 \rightarrow e_1 R\, e_2$.

We now demand a relation R suitable for building meta-levels to be:

acyclic $\forall\, e_1, e_2, n : e_1 R^n\, e_2 \rightarrow \neg\, e_2 R\, e_1$, and

anti-transitive $\forall\, n \geq 2 : R^n \cap R = \emptyset$.

Note that "acyclic" implies both "irreflexive" ($\neg \exists\, e : e R\, e$) and "asymmetric" ($\forall\, e_1, e_2 : e_1 R\, e_2 \rightarrow \neg\, e_2 R\, e_1$), but extends the exclusion of cycles above length two as well.

Considering "model instantiation" (e.g., $\lhd_t$) as a candidate for a meta-level constructing relation, we can confirm that it should not map elements to themselves[8], should not claim that elements are mutual instances of each other (creating circular definitions), and finally should disallow transitivity of any length. This is why our "anti-transitive" property excludes transitivity not just for two levels, but for any chain-length with $n \geq 2$.

The above constraints guarantee a (meta-) level-constructing relationship, however, relationships might be "loose" in the sense that they may cross more than one level boundary. If this is undesired, as it is for strict metamodeling [16], a further property is required:

level respecting $\forall\, n, m :$
$(\exists\, e_1, e_2 : e_1 R^n e_2 \wedge\ e_1 R^m e_2) \rightarrow$
$n = m$

Using this novel, purposed designed property for relations, we can make sure that all paths from one element e_1 to another element e_2 have the same lengths. Note, that unique paths are unproblematic anyway as they assign levels to their involved elements "by definition" without any possibility of inconsistencies. Multiple paths, however, may exist due to multiple

[8] Self-description is useful for self-terminating meta-hierarchy tops, however. This rather special application, which makes sense for linguistic hierarchies (see section 4.1) only, can be admitted as a special case.

classification. Note that property "level-respecting" implies property "anti-transitive", which then might be discarded.

Going back to our initial question we can now firmly reject any potential "metamodel status" of the right hand side model of Fig. 1, since the relationship $\lhd$ between the models is actually the "token-model-of" ($\lhd_i$) relationship, which is transitive and therefore not suitable to constitute metamodels. Hence, a token model of a token model is not a metamodel.

Fig. 4 shows a model (at the right hand side) which is truly a metamodel.[9] This time the relationship between the models is "type-model-of" ($\lhd_t$) and therefore the litmus test, whether the relationship is not transitive, succeeds. As a result, we can confirm that the phrase "A metamodel is a model of a model" is true, provided that the respective "model-of" relationship is not transitive.

Note that in order to create a metamodel we need the *same* non-transitive relationship (e.g., "type-model-of") twice. Even given that in Fig. 3 "Java Classes" is a type model of "Java Runtime" and "Token & Type Model" is a (token) model of "Java Classes", this does not make the latter a metamodel of "Java Runtime", as we are missing another "type model" relationship.

4.1 Flavors of Model and Element Instantiation

We have seen that we can construct a model $\mathcal{M}_2$ of another model $\mathcal{M}_1$ so that $\mathcal{M}_2$ is not a model of $\mathcal{M}_1$'s subject $\mathcal{S}$: $\mathcal{S} \lhd_t^2 \mathcal{M}_2 \nrightarrow \mathcal{S} \lhd_t \mathcal{M}_2$. An alternative way, to achieve such anti-transitivity, potentially giving rise to metamodels, is not to model the content of model $\mathcal{M}_1$, but the language that was used to write model $\mathcal{M}_1$. The model on the right hand side of Fig. 5 (at L_1) specifies (the abstract syntax of) the language used to create the token and type models, in this case a tiny portion of the UML, which is why the respective "instanceOf" relationships are labeled with $\lhd_t^l$ ("*l*" for linguistic).

In order to be able to discuss (in section 4.2) which of the models in Fig. 6 might be granted metamodel status and to fully appreciate the discussion (in section 5) on whether or not it is reasonable to associate models with language definitions, we need to make the difference between linguistic ($\lhd_t^l$) and ontological ($\lhd_t^o$) instantiation [17] explicit. Note that while it is possible to distinguish between the instantiation relationship between models (called "sem" in [1]) and the (inter-level) instantiation relationship between model elements (called "meta" in [1]) we will just use one overloaded term for both cases.

What does it mean for a model (-element) to be an instance of another model (-element)? Fig. 5 depicts that the answer depends on whether one is talking about ontological or linguistic instantiation.

Ontological instantiation can be defined as

$$e \lhd_t^o T \rightleftharpoons \mu(e) \in \varepsilon(\mu(T)), \text{or alternatively} \tag{7}$$
$$e \lhd_t^o T \rightleftharpoons \iota(\mu(T))(\mu(e)). \tag{8}$$

[9] We could have also extended Fig. 2 to include yet another type model, e.g., containing elements "LocationType" & "ConnectorType", but Fig. 4 shows a nice natural name for a metatype ("Breed").

See the left hand side of Fig. 5 for a corresponding visualization[10], which uses real images for denoting the *meaning* of models for illustrative purposes only.

For an object to be considered an ontological instance of a type, we expect its referenced element to be in the extension of the concept referenced by the type (definition 7). The intensional variant (definition 8) demands that the referenced domain element satisfies the intension (a conjunction of predicates) of the referenced concept.

Ontological instantiation between two elements or models is therefore based on the relationship between them in terms of their meaning. For ontological domain models we may set this meaning to be the corresponding elements in the reduced system $\mathcal{S}_r$, which in turn reference elements in the original system $\mathcal{S}$, i.e. $\mu(\mathcal{M}) = \pi(\mathcal{S})$. This way we can distinguish between the original system that is represented (ρ) by the model and—a possibly much less rich—meaning of the model (μ) (see also Fig. 7).

Note that practical modeling languages, like UML, additionally define syntactic conformance rules between e and T (see equation 7). This conformance is based on whether or not e can be regarded as an instance of T syntactically. This makes sense, as the real ontological test can not be performed automatically.

Linguistic instantiation can be defined as

$$e \lhd_t^l T \rightleftharpoons e \in \varepsilon(\mu(T)), \text{or alternatively} \tag{9}$$
$$e \lhd_t^l T \rightleftharpoons \iota(\mu(T))(e). \tag{10}$$

See the bottom part of Fig. 5 for a corresponding visualization. Linguistic instantiation between an element and a linguistic type is based on the assumption that the type represents a (fragment of a) language defining which expressions are valid sentences of it. Therefore, in definition 10 we simply apply the intension of the type—a predicate—to the element. Note that the element appears as itself, instead of being an argument for μ, since linguistic instantiation concerns the *form* of elements themselves, as opposed to their *content* (and *meaning* respectively) as is the case with ontological instantiation.

As can be seen in Fig. 5, language concepts, such as "Object" and "Class" on the right hand side, do not reference any subjects from the domain on the left hand side. This illustrates that a linguistic model $\mathcal{M}_l$ is never a model of its subject model's subject:

$$\mathcal{S} \lhd_i \mathcal{M} \wedge \mathcal{M} \lhd_t^l \mathcal{M}_l \rightarrow \neg(\mathcal{S} \lhd_t \mathcal{M}_l)\text{, whereas} \tag{11}$$
$$\mathcal{S} \lhd_i \mathcal{M} \wedge \mathcal{M} \lhd_t^o \mathcal{M}_t \rightarrow \mathcal{S} \lhd_t^o \mathcal{M}_t \tag{12}$$

4.2 Is the UML Metamodel a Metamodel?

Fig. 6 shows one model (bottom-left) depicted by an object diagram and one corresponding ontological type model depicted as a class diagram (top-left). For both, the corresponding excerpts from the linguistic UML metamodel (here, unconnected), are shown on the right hand side. Which of these

[10] For the sake of simplicity we are just considering two ontological levels, whereas in principle there could be n.

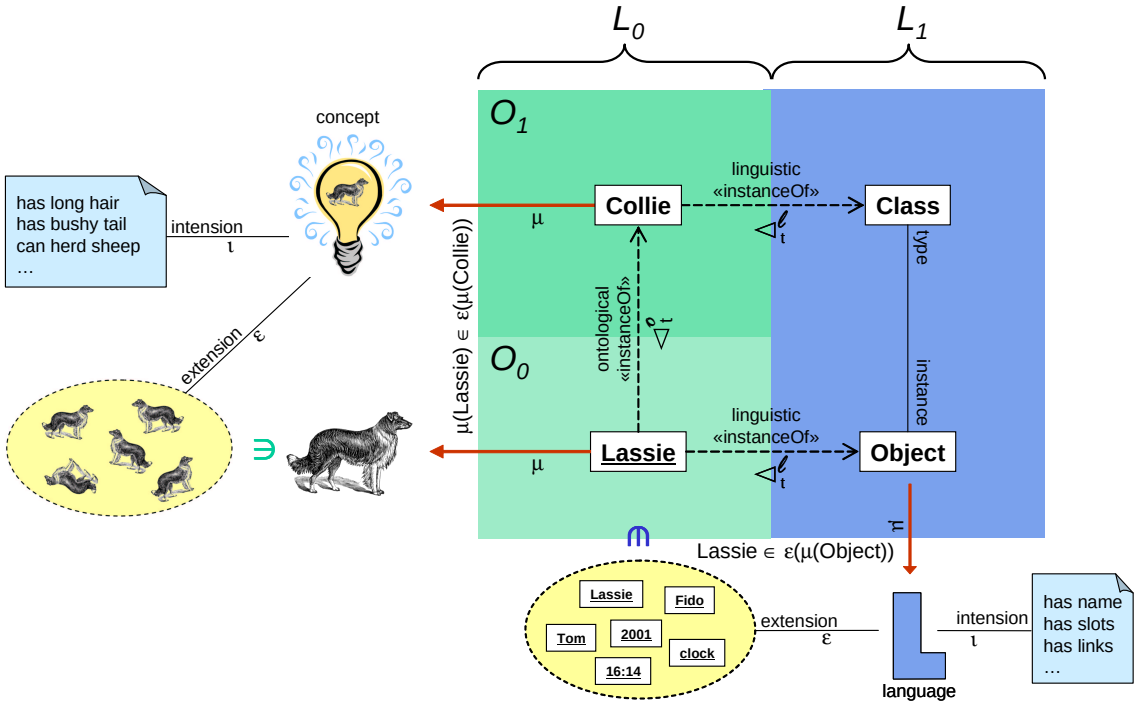

Fig. 5 Ontological versus Linguistic Instantiation

four models/model-fragments can be granted metamodel status?

The top-left type model is not a metamodel with respect to the subject $\mathcal{S}$ (not shown in Fig. 6) of the bottom-left model. According to the test we have developed at the beginning of section 4, we require a sequence of two non-transitive "model-of" relationships, but we only have a single "type model-of" relationship, making the top-left model a simple type model of $\mathcal{S}$ (see equation 12).

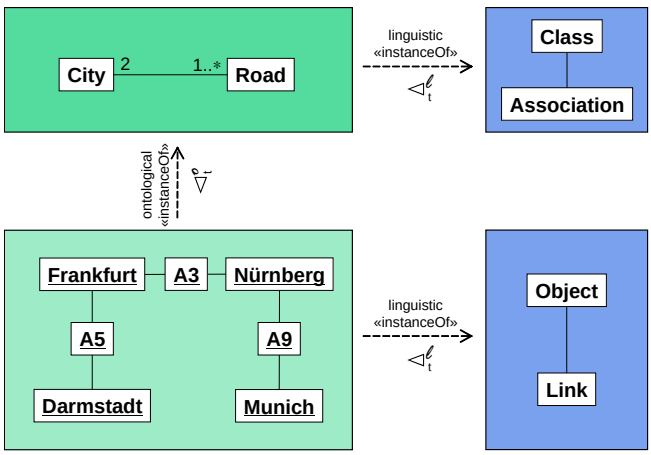

Fig. 6 Linguistic Metamodels

The interesting cases are the two models at the right hand side of Fig. 6. The bottom-right model is not a result of applying the same non-transitiv "model-of" operation twice to system $\mathcal{S}$, but might still be called a metamodel on the basis that it is a model of a model, without also being a model of the bottom-left model's subject $\mathcal{S}$ (see definition 11). Hence, even though it does not maintain an R^2 relationship to $\mathcal{S}$, it nevertheless, engages in an overall anti-transitive relationship.

The reason why we are discussing this "unclean" case of a metamodel is the OMG's policy of referring to the bottom-right model as a metamodel [15]. In fact, for old versions of the four-layer architecture (where objects like "Frankfurt" were located at the M_0 level, classes like "City" at the M_1 level and elements of the UML superstructure like "Class" at the M_2 level) this seemed plausible as the language level M_2 could be thought of maintaining a two-level "type-model-of" relationship ($\lhd_t^2$) with user objects at level M_0. Yet, with the change (correction) of the interpretation of M_0 as not belonging to the model stack [1], user model objects moved to M_1 and now are only in one-level type-model-of relationship with the UML language definition at M_2 ($\lhd_t$). One might think that there is still a two-level "type-model-of" chain to the real user individuals at M_0, but this is not the case as the relationship between M_1 and M_0 concerning individuals is not "type-of" but "represents" (ρ). Consequently, there is no clean case of a two-level type-model chain.

What about the top-right model of Fig. 6? Superficially it appears to feature a two-level type-model chain, as there are two $\lhd_t$ relationships from the bottom-left to the top-right model. However, the two $\lhd$ relationships are not of the same kind. Ontological instantiation ($\lhd_t^o$) relates two models whose subjects are in the same domain but on different logical levels. Linguistic instantiation ($\lhd_t^l$) relates a model with the definition of the language of which it is an expression. Ergo, the top-right model is on the same linguistic level as

notation	*name*	*description*
α	abstraction	creates a model from a system using projection (π) and possibly classification (Λ) and generalization (Γ), hence $\mathcal{S} \lhd \alpha(\mathcal{S})$.
Λ	classification	creates a type model, hence $\mathcal{M} \lhd_t \Lambda(\mathcal{M})$
Γ	generalization	creates a supermodel, hence $\mathcal{S} \lhd_t \mathcal{M} \rightarrow \mathcal{S} \lhd_t \Gamma(\mathcal{M})$
π	projection	a homomorphic mapping creating a reduced system from a given system, using selection and reduction of information.
ρ	represents	records the intention of a model to represent a system.
μ	meaning	assigns meaning to a model (element). If $\rho(\mathcal{S}, \mathcal{M})$ then one may define $\mu(\mathcal{M}) = \pi(\mathcal{S})$.
$\lhd$	model-of	holds between a system and a model describing the former.
$\lhd_i$	token model-of	holds between a system and a model representing the former in a one-to-one fashion. Model elements may be regarded as designators for system elements.
$\lhd_t$	type model-of	holds between a system and a model classifying the former in a many-to-one fashion. Model elements are regarded as classifiers for system elements.
$\lhd^o$	ontological model-of	indicates that the model controls the *content* of its elements, hence $\mathcal{S} \lhd_t^o \mathcal{M} \rightleftharpoons \mu(\mathcal{S}) \in \varepsilon(\mu(\mathcal{M}))$ and $\mathcal{S} \lhd_i^o \mathcal{M} \rightleftharpoons \mu(\mathcal{M}) = \pi(\mu(\mathcal{S}))$. Assuming $\mu(\mathcal{S}) = \mathcal{S}$, for systems which do not model anything, we have $\mathcal{S} \lhd_i^o \mathcal{M} \rightleftharpoons \mu(\mathcal{M}) = \pi(\mathcal{S})$ and thus $\rho(\mathcal{S}, \mathcal{M}) \rightarrow \mathcal{S} \lhd_i^o \mathcal{M}$ (see definition of μ above).
$\lhd^l$	linguistic model-of	indicates that the model controls the *form* of its elements. This automatically implies $\lhd_t^l$ and, hence $\mathcal{S} \lhd_t^l \mathcal{M} \rightleftharpoons \mathcal{S} \in \varepsilon(\mu(\mathcal{M}))$

Table 2 Notation Overview

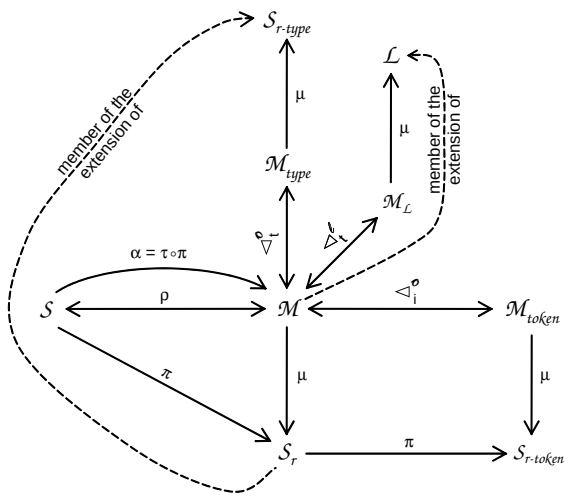

Fig. 7 Relations and Functions

the bottom right model, because the intermediate ontological instantiation does not count with respect to linguistic metalevels.

Therefore, strictly speaking, neither of the two right hand side models of Fig. 6 are pure metamodels in the sense of repeating the same non-transitive operation twice. If one wants to stick to the term "metamodel" for level M_2 (and the right hand side models of Fig. 6 respectively) one has to do so on the basis that linguistic models are models of models without violating anti-transitivity.

Fig. 7 and Tab. 2 summarize the concepts and notation introduced so far.[11] We will make use of these in the next section in order to finally explore whether the use of "model" is adequate in the case of linguistic models, and if so, why.

5 Metamodel = Language Definition?

The first task in answering this question is to make the question more precise: With respect to *ontological* metamodels (e.g., the rightmost model of Fig. 4) we can observe that their *primary* purpose is *not* to define a language. Albeit this is not how most users think about their domain models, one may still regard them as defining a vocabulary plus constraints to be used in the next lower ontological level, just as a class diagram can be regarded to define a vocabulary plus constraints, i.e., a language, for all object diagrams conforming to it. This is not surprising as the definitions for ontological and linguistic instantiation (definitions 8 and 10 respectively) only differ in the "detour" via the domain subjects in the ontological case. However, for all intents and purposes one can still answer the question whether *ontological* metamodels are language definitions with "no", unless we interpret an ontological metamodel as a domain specific language definition, thus turning it into a linguistic metamodel.

So let us concentrate on the question what the relationship between *linguistic* metamodels and languages is. A few

[11] Fig. 7 does not use the UML notation but indicates relations with lines featuring arrow heads on both sides and functions with lines featuring one arrow head only; the two dashed lines being exceptions to this rule.

quotations indicate that there is at least a perceived, close relationship:

> "[A metamodel is a] model that defines the language for expressing a model [15]."

> "A metamodel is a model of a language of models [2]."

> "A metamodel is a specification model for which the systems under study being specified are models in a certain modeling language [10]."

Reconstructing these statements in our framework yields: A metamodel $\mathcal{MM}$ is the model of a model $\mathcal{M}$, i.e., $\mathcal{M} \lhd_t^l \mathcal{MM}$, if $\mathcal{M} \in \varepsilon(\mu(MM))$ or equivalently, making the language involved explicit: $\mathcal{M} \in \varepsilon(\mathcal{L}) \wedge \mathcal{L} = \mu(MM)$ (see Fig. 8 for one language and Fig. 9 for a language stack). We use a language concept $\mathcal{L}$ and refer to the language specification with $\iota(\mathcal{L})$, and to the set of all sentences of $\mathcal{L}$ with $\varepsilon(\mathcal{L})$.

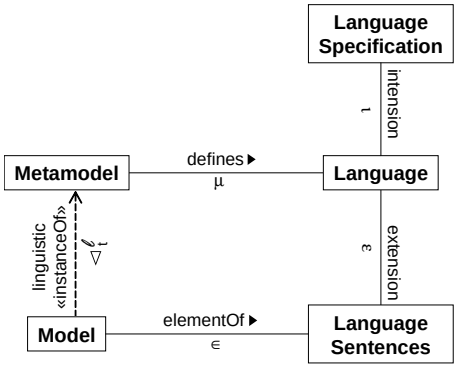

Fig. 8 Metamodels as Language Definitions

As a result, a model is an instance of a metamodel if it is an element of the set of all sentences which can be generated with the language associated with the metamodel (also see definition 9). Note that in formal treatments the term "language" is often associated with the said set of all language sentences (labeled "Language Sentences" in Fig. 8). In practice, this language extension is almost always defined by an intensional definition, i.e., rules characterizing whether or not an expression is a sentence of the language. Hence, we also could have interpreted the metamodel to be directly one of many equivalent language specifications, i.e., $\mathcal{MM} \sim \iota(\mathcal{L})$. Our choice presented above, however, has the advantage of being more symmetric in comparison to ontological instantiation.

Let us refocus on the initial question of whether the term "model" is appropriate for a language specification, such as the M_2 layer of the OMG's four-layer architecture. Surely we should not use the term "model" simply because the specification was expressed using a modeling language. For a true model we would still expect some reduction feature, which is in conflict with the expectation that a language specification should be precise and complete.

One might argue that just the abstract syntax of the language is defined by a metamodel and other aspects belonging

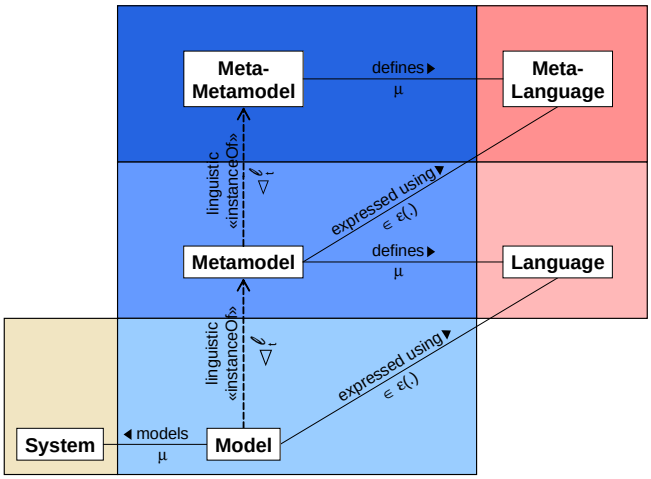

Fig. 9 Language Definition Stack

to a complete language specification, such as concrete syntax and semantics are left out. But would we want to stop using the term "metamodel" if these aspects were somehow included in the future?

Fortunately, we do not need to engage in a discussion about the existence of a reduction feature with respect to the *language specification*. To justify the model nature of a language specification, expressed through a so-called metamodel, it is sufficient to recognize that it universally captures all models that may be expressed with it, i.e, are instances of it. Hence, it is its capacity as a *type model* for all the models expressible with it—as opposed to its capacity as a *token model* for the language specification—that qualifies it as a model. A language metamodel therefore does not deserve its name for what it means (a language), but for what it classifies (linguistic instances of it).

6 Related Work

Bézivin [5] and Favre [2] recommend "conformantTo" and "ConformsTo" over "instanceOf" in the context of relating models to each other in order to distinguish the conformance relationship between models from the instantiation relationship known from object-orientation (i.e., between objects and classes). For better or for worse, however, "instanceOf" is already a widely used term [15] for relating models to each other, and it appears justifiable to overload the term in this way, given the analogy between the type-model/token-model and class/object pairs respectively.

Be that as it may, in the context of a description hierarchy such as the OMG four-layer architecture, there are good reasons to have different names for inter-level and intra-level relationships between elements. Any combination making the difference explicit seems to be acceptable (see Tab. 3 for a comparison of terminology and section 4.1 for a corresponding discussion).

Strahringer also takes a systematic look at how description hierarchies are constructed and coins the term "metaiza-

	intra-level	*inter-level*
[1]	instOf	meta
[5]	instanceOf	conformantTo
[15]	snapshot	instanceOf
[17]	ontological instanceOf	linguistic instanceOf

Table 3 Instantiation Terminology for the OMG Stack

tion principle[12]" for the operation that is repeatedly applied from level to level [18]. She also points out that counting meta levels, e.g., in order to ascertain whether a metamodeling level has been reached, has to be done with respect to one "metaization principle" only, in case several are employed in a description chain. She does not, however, formally define the requirements for a metaization principle which allows the construction of a meta-hierarchy.

Strahringer's analysis of the relationship between models and languages [18] is similar to ours (see Fig. 9), however, using a different distribution of elements to levels and a different terminology.

Seidewitz distinguishes two kinds of meaning for models [10]: He describes an *interpretation* ("meaning in the first sense") of a model

> "...as a mapping of elements of the model to elements of the SUS[13]... [i.e., for instance, this] class model means that the Java program must contain these classes."

He describes the *theory* of a modeling language ("meaning in the second sense") as

> "the relationship of a given model to other models derivable from it. ...[i.e., for instance, this] class model means that instances of these classes are related in this way."

Seidewitz's two kinds of meaning may hence be explained in our terminology by considering whether the model in question is an ontological model or a linguistic model. The "meaning" of an ontological model (*interpretation*) relates (horizontally) to the domain of interest (through μ and ρ). The "meaning" of a linguistic model (*theory of a modeling language*) relates (vertically) to the next metalevel below, enabling other models to be checked against the linguistic model for conformance (through the "$\in \varepsilon(\mu(\mathcal{MM}))$" check).

Seidewitz's recognition of both *interpretation* and *theory of a modeling language* as being relevant for models in general is also matched by our observation that a model with an ontological intention can always also be used with a linguistic interpretation in order to provide a syntactic conformance check for subjects of which it is a type model.

Favre also defines a function μ relating a model to the system it represents [2]. Since "represents", in the sense of "could be regarded as a model for" is not a many-to-one, i.e., functional, relation, the author assumes that Favre also means $\mu(\mathcal{M})$ to refer to a single associated meaning of $\mathcal{M}$.

Furthermore, Favre defines a "meta-step" pattern, which is similar to the characterization of linguistic instantiation presented here. According to Favre, a model $\mathcal{M}$ conforms to a metamodel $\mathcal{MM}$, if it is an element of the language represented by the metamodel, i.e., (using our notation)

$$\mathcal{M} \lhd_t^l \mathcal{MM} \rightleftharpoons \mathcal{M} \in \varepsilon(\mathcal{L}) \wedge \varepsilon(\mathcal{L}) = \mu(\mathcal{MM})^{14}.$$

Favre, thus interprets the metamodel to directly represent all language sentences $\varepsilon(\mathcal{L})$. In contrast, the approach presented here

$$\mathcal{M} \lhd_t^l \mathcal{MM} \rightleftharpoons \mathcal{M} \in \varepsilon(\mathcal{L}) \wedge \mathcal{L} = \mu(\mathcal{MM}),$$

assumes linguistic models to represent language concepts which in turn have an extension (set of all language sentences) and an intension (a language specification).

7 Conclusion

In order to establish a commonly agreed terminology it is essential for the model driven engineering community to define under which circumstances the notions "model" and "metamodel" and its associated basic relationships are applicable. This article argued for maintaining the required features already known for technical models and refrain from overly extending the notion of "model", e.g., to include "copies".

We used an approach where the subjects of modeling are already available in a finite representation. We have intentionally ignored the process of capturing systems from the real world into a representation, as this necessarily implies a number of abstraction operations which then are no longer optional. The approach used in this article made it possible to discuss abstraction functions with varying reduction degrees (including the extreme case of no reduction at all, i.e., exact copying), something simply impossible when dealing with real world subjects.

In order to be able to judge under which circumstances a model might be granted "metamodel" status, it was extremely helpful to distinguish between token model and type model roles. Without such a means of discrimination, a discussion about statements like "A metamodel is a model of a model" cannot be settled systematically.

We have introduced a systematic definition, based on *acyclic* and (the novel notion of) *level-respecting* relations, to decide under which circumstances a model maybe granted "metamodel" status. It became apparent that the OMG's policy of referring to the UML language definition as the "UML metamodel" no longer has a straightforward justification with respect to the latest version of the four-layer architecture, but can be justified to allow a consistent interpretation based on anti-transitivity of model relationships.

[12] In German: "Metaisierungsprinzip".
[13] System under study.
[14] In this context, another useful condition made by Favre, $\mu(\mathcal{M}) = \mathcal{S}$, is not important.

We used the terms "role" and "reference mode" to emphasize the fact that whether a model is a token or a type model depends on its relationship to its subject. This is an important observation as we have seen examples of models being both a token and a type model at the same time (with respect to different subjects).

The reference mode "token model" clearly demonstrates that the "represents" relationship from a model to a system does not correspond to "instanceOf" from object-technology. It turns out to be wrong to interpret systems to be instances of their token models. While the reduction feature of token models may sometimes create the impression that classification occurred, really only representation takes place.

We have, furthermore, shown that "generalization" is also a basic relationship between models in addition to "instantiation" and "representation". Again, the distinction between token and type models significantly simplified the analysis of the interplay of the generalization relationship with the other basic relationships.

Finally, we argued that language definitions may rightfully be referred to as (meta-)*models* regarding their type model capacity as opposed to their token model capacity.

The author believes that the recognition of token and type model roles and an explicit treatment of all basic notions in modeling, including "generalization", may drastically simplify disputes about fundamental issues, such as the "metamodel" definition, and will provide a useful basis to build on.

Acknowledgements The author would like to thank the participants of the Dagstuhl seminar 04101 on "Model-Driven Language Engineering" for many stimulating discussion. In particular (in alphabetical order) Pieter van Gorp, Martin Grosse-Rhode, Reiko Heckel, and Tom Mens further contributed by sending emails to the author with their views on what modeling is about. Discussions with Colin Atkinson and Friedrich Steimann led to insights which motivated and influenced this article. Finally, I'm grateful for many comments contributed by Wolfgang Hesse and the commitment of the anonymous reviewers which led to a number of significant improvements.

References

1. Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proceedings of the 16th International Conference on Automated Software Engineering Coronado Island*, pages 273–280, November 2001. 1, 3, 9, 10, 13
2. Jean-Marie Favre. Towards a basic theory to model driven engineering. In *Third Workshop in Software Model Engineering (WiSME@UML)*, October 2004. 1, 3, 12, 13
3. Chen-Chung Chang. *Model Theory*. North-Holland, Amsterdam, 2 edition, 1977. 1
4. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Mass., 1999. 1, 2
5. Jean Bézivin. In search of a basic principle for model driven engineering. *Special Novática Issue "UML and Model Engineering"*, V(2), April 2004. 1, 7, 12, 13
6. Roland Kaschek. A little theory of abstraction. In Bernhard Rumpe and Wolfgang Hesse, editors, *Modellierung 2004, Proceedings zur Tagung, 23.-26. März 2004, Marburg*, volume 45 of *LNI*, pages 75–92. GI, 2004. 2
7. Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, Wien and New York, 1973. 2
8. Richard S. Bird. An Introduction to the Theory of Lists. Technical Report PRG-56, Oxford University, October 1986. 3
9. W. Steinmüller. *Informationstechnologie und Gesellschaft: Einführung in die Angewandte Informatik*. Wissenschaftliche Buchgesellschaft, Darmstadt, 1993. 3
10. Ed Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, September 2003. 3, 12, 13
11. Michael Harkavy and et al., editors. *Webster's new encyclopedic dictionary*. Black Dog & Leventhal publishers Inc., 151 West 19th Street, New York 10011, 1994. 3, 4
12. Rudolf Carnap. *Meaning and Necessity: A Study in Semantics and Modal Logic*. University of Chicago Press, 1947. 5
13. Jochen Ludewig. Models in software engineering—an introduction. *Journal on Software and Systems Modeling*, 2(1):5–14, March 2003. 5
14. OMG. *MDA Guide Version 1.0.1*, 2003. Version 1.0.1, OMG document omg/03-06-01. 8
15. OMG. *Unified Modeling Language Infrastructure Specification, Version 2.0*, 2004. Version 2.0, OMG document ptc/03-09-15. 8, 10, 12, 13
16. Colin Atkinson and Thomas Kühne. Profiles in a strict metamodeling framework. *Journal of the Science of Computer Programming*, 44(1):5–22, July 2002. 8
17. Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, September 2003. 9, 13
18. Susanne Strahringer. *Metamodellierung als Instrument des Methodenvergleichs*. Shaker Verlag, Aachen, 1996. 13