Subjects, Models, Languages, Transformations

Arend Rensink

Department of Computer Science, University of Twente
P.O.Box 217, 7500 AE, The Netherlands
rensink@cs.utwente.nl

Abstract. Discussions about model-driven approaches tend to be hampered by terminological confusion. This is at least partially caused by a lack of formal precision in defining the basic concepts, including that of "model" and "thing being modelled" — which we call *subject* in this paper. We propose a minimal criterion that a model should fulfill: essentially, it should come equipped with a clear and unambiguous *membership test*; in other words, a notion of which subjects it models. We then go on to discuss a certain class of models of models that we call *languages*, which apart from defining their own membership test also determine membership of their members. Finally, we introduce *transformations* on each of these layers: a subject transformation is essentially a pair of subjects, a model transformation is both a pair of models and a model of pairs (namely, subject transformations), and a language transformation is both a pair of languages and a language of model transformations. We argue that our framework has the benefits of formal precision (there can be no doubt about whether something satifies our criteria for being a model, a language or a transformation) and minimality (it is hard to imagine a case of modelling or transformation not having the characterstics that we propose).

1 Introduction

The literature on model-driven architecture (MDA, see [16]) is dominated by the picture of a four-layer modeling hierarchy, in which each layer is populated by instances of the model one level higher, and the sole inhabitant of the top layer can be seen as an instance of itself. This certainly makes for a very elegant structure; however, too little attention is paid to a precise, unambiguous definition of the building blocks in that structure. The most glaring omission is that of *instantiation*: what exactly is meant by "something is an instance of something else"? This omission has been pointed out by others before (e.g., [2,6]); a careful investigation has led several researchers independently to the conclusion that there are at least two different, incompatible kinds of instantiation involved.

We take a different tack: rather than fixing, or trying to fix, a particular notion of model and instantiation, we take a closer look at the relative *roles* of a model and the things it models. That is, we do not follow the idea that there should be four absolute layers of modeling with a global notion of instantiation; rather, we allow the introduction of model/instance relations that may occur

Dagstuhl Seminar Proceedings 04101
http://drops.dagstuhl.de/opus/volltexte/2005/24

everywhere; the only requirement is that a model carries with it a clear and unambiguous notion of what is an instance and what is not. We then proceed by extending this minimal core with the concepts of *language*, which we define as a special kind of model of models, and *transformation*, which we interpret in terms of binary relations over models.

The purpose of this work is twofold:

- Reduce terminological confusion. We have observed that many discussions on model-driven engineering are frustrated by the fact that people hold different interpretations of the concepts of modeling and meta-modeling; sometimes without even realizing it, or sometimes (even worse) while insisting that their interpretation is *the* (only) correct one. We try to remedy this by giving formal criteria (using only elementary algebra) for a very small vocabulary; we argue that this can be used, on the one hand, as an underpinning of existing work, and on the other hand, to show up subtle distinctions between existing interpretations.
- Focus on core ideas. We believe that the concept of model-driven engineering is very worthwhile and enlightening, and we subscribe to the OMG's vision regarding its use; but we do *not* believe that it should be universally applied or carried on *ad infinitum*. Rather, one should keep the purpose of a given model in mind, and on a case-by-case basis investigate the usefulness of formulating a higher-level model of which the first one is an instance. For similar reasons, we see little benefit in an absolute numbering of levels of meta-modeling. Instead, we present the smallest set of concepts that, in our opinion, encompasses the required *relation* between models and their members.

Let us start by laying down some of the terminology that we will be using.

Subjects. This is what we call every entity *in the domain of discourse*. That is, in any given application one has to delimit that domain, in other words, one has to make clear what subjects are involved in that application. By this requirement we avoid theoretical pitfalls such as Russel's paradox.

Models. These are subjects with one special feature, namely a *membership test*. Essentially, the membership test is a function stating, for every subject, whether it is or is not a member of the model in question.

Elementary though it is, the above already embodies some distinguishing characteristics. In particular, we reject the idea that there is a universal notion of instantiation, that is the same over all models. Rather, our membership test comes with the model. Thus, in our view, it is quite acceptable that given subject is a member of more than one model.

Languages. These are special kinds of models of models with one additional feature besides their membership test, namely a *correctness criterion*. The correctness criterion induces the membership test of all members. That is, not only does a language have a way to recognize its own members (through its own membership test, which it has by virtue of the fact that it is a model), but also to establish the members of its members.

Transformations. These are essentially ordered pairs of subjects: every transformation identifies *left* and a *right* element. Presumably one is obtained by manipulating the other, but we do not enforce this in any way. Note that transformations are themselves also subjects, and may therefore be members of models; moreover, the pair of subjects in a given transformation may be models.

In fact, the "modeling dimension" and the "transformation dimension" are orthogonal: we may unambiguously speak of transformation models/languages or model/language transformations, and even of model transformation models and model transformation languages, and so on.

The ideas outlined above are made more precise in two ways. Firstly, we provide some (elementary) mathematical definitions for the core concepts of subjects, models, languages and transformations; and we also give an equivalent pictorial representation in the form of a number of UML-like class diagrams. Secondly, we give a range of examples intended to clarify by illustration.

This work originated at the Dagstuhl seminar on "Language Engineering for Model-Driven Software Development" (see [4]); this paper presents the current, still preliminary, state of the framework. This owes much to the other workshop participants, in particular Colin Atkinson, Ralf Laemmel, Daniel Moldt and Reiko Heckel.

2 Models

All the definitions below require a *universe* of subjects `Subject`. This means that the set of subjects, which is the domain of discourse for a given application of the framework, must be given. It does not need to be finite and it does not have to be a mathematical set. $\mathbb{B}$ denotes the set of boolean values $\{\mathbf{tt}, \mathbf{ff}\}$.

Definition 2.1 (models). *Given a universe* `Subject`*, a* model *is a subject* $M \in$ `Subject` *with an associated boolean function* $\texttt{isMember}_M\colon \texttt{Subject} \to \mathbb{B}$ *stating, for every subject in* `Subject`*, whether it satisfies (is a member of) the model.*

As a shorthand, we write $s \models M$ for $\texttt{isMember}_M(s)$. From a mathematical point of view, the above definition may seem overly complex: a boolean function such as `isMember` is equivalent to a unary predicate over `Subject`, which in turn is equivalent to a subset of `Subject`. We have chosen this formulation for several reasons:

- It is closer to the intuition of most software engineers;
- It emphasizes that the membership function is intended as intensional and not extensional, by which we mean that a model does not have to be able to enumerate or generate all its members;
- It opens the way to generalizations where this function may be partial (reflecting that membership may be undecidable) or range over the fuzzy booleans (reflecting that there may be uncertainty about membership). We do not, however, investigate such generalizations in the current paper.

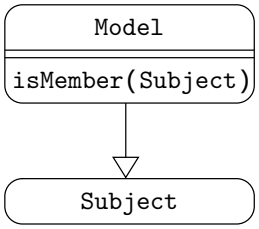

Fig. 1: UML diagram for the concepts in Definition 2.1

In UML notation, the framework so far is depicted in Figure 1. For the same reasons as given above, we have deliberately chosen not to represent the membership function as an association between `Model` and `Subject`.

Examples. We give a series of examples to show the utility of this definition and to strengthen the intuition.

1. Every type (in the programming language sense of the word) serves as a model for the values of that type. The membership test can be specified for instance by typing judgments of the form $\Gamma \vdash f : T$, where T is a type, f a value (i.e., a term) of that type, and Γ a context specifying types for the free variables possibly occurring in f. The membership test is implemented by the type checking algorithm.
2. Similarly, every database scheme is a model for all databases set up according to that scheme; the membership function is implicit in the semantics of the database scheme.
3. XML documents can be modeled in two different ways: either through an XML Data Type Definition or through an XML Schema. Both are model in out sense; the membership function is implicit in the semantics of DTD's, resp. schemas. (This example shows that a given subject can very well have more than one model.)
4. The grammar of a programming language such as Java is a model for all Java programs. The membership function is the parsing function.
5. Every program is a model for its actual executions. The membership function is trivial, since an execution is a volatile thing which can only be obtained from the program in the first place, viz., by executing it.
6. Every class diagram can be seen as a model for all object diagrams that contain instances of the classes of the diagram, with associations and attributes also as specified by the class diagram. The membership function is a many-to-one relation between the nodes in the object diagram (i.e., the objects) and the nodes in the class diagram (i.e., the classes) that preserves the associations and attributes; this can for instance be formalized through a morphism between the diagrams.
7. Another interpretation of a class diagram is as a model of program states, namely those states that (on the heap) have instances of the classes shown in the diagram. The membership function can be defined through serialization and reflection.

8. The UML meta-model can serve as a model for all types of UML diagrams: class diagrams, sequence diagrams, state diagrams and so on. Each type of diagram corresponds to a different membership function. In our terms, therefore, the UML meta-model is not yet a model but becomes one when a specific membership function is also provided. For instance, the membership function for class diagrams states that diagrams may are classes, operations, associations and so on, with corresponding constraints, but the concept of state (which is in the meta-model) plays no role in this context. Note that these membership functions have not been formalized in the UML standard; we consider this to be a real omission.
9. The *Meta Object Facility* caps off the OMG's meta-level hierarchy; it is seen as a model for all types of meta-model, such as the UML and XML meta-models, and also as a model for itself. A membership function that supports this view might be defined along the following lines: in order to be a member of the MOF, a given subject has to explicitly provide its own interpretation of the concepts present in the MOF; that is, it has to state what its model elements, types, classifiers etc. are, and also show that those elements satisfy the required constraints. Thus, the responsibility for showing membership is put largely on the side of the would-be member, rather than on the side of the model.
10. Every scientific model of a real-world system is a model in the sense of Definition 2.1. The membership function is determined through experimentation, viz. by testing that the system behaves as predicted by the model. (Actually, this is an example where the membership function is not really to the booleans, but to a fuzzy or probabilistic space there it can typically not be established with absolute certainty that the model describes the real-world system in all circumstances.)

3 Languages

Having a model does not make a method model-driven. Rather, it is the ability to systematically change, adapt or evolve the model, with well-defined consequences in terms of the subjects being modeled, that lies at the heart of the model-driven philosophy. Such systematic changes require that the model itself be the subject of another model. Thus, we require a model of models. This is usually called a *meta-model* in the literature. In our vision, however, it is not enough to have a meta-model; in addition to that, we require the aforementioned "well-defined consequences in terms of the subjects being modeled." Concretely, we want meta-models with the additional capability to reason about the members of their members. We will use the term *language* for such a special meta-models. The required additional capability of languages is defined as follows.

Definition 3.1 (languages). *Given a universe* $\texttt{Subject}$ *and a set* $\texttt{Model} \subseteq \texttt{Subject}$*, a* language *is a model* $L \in \texttt{Model}$ *with an associated boolean function* $\texttt{isCorrect}_L\colon(\texttt{Model} \times \texttt{Subject}) \to \mathbb{B}$ *stating, for every model* M *and subject* s*,*

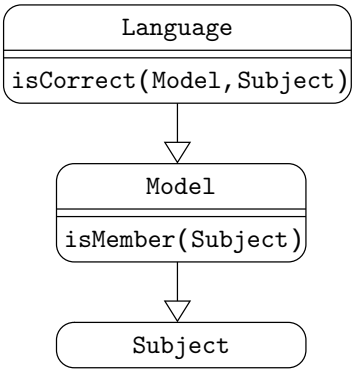

Fig. 2: UML diagram for the concepts in Definition 3.1

whether s is a member of M according to L. The following equivalence should hold for all $M \in$ `Model` and $s \in$ `Subject`:

$$\texttt{isCorrect}_L(M,s) \iff \texttt{isMember}_L(M) \wedge \texttt{isMember}_M(s) \ . \qquad (1)$$

We write $s \models_L M$ for $\texttt{isCorrect}_L(M,s)$. Again, we have chosen to represent $\texttt{isCorrect}_L$ as a boolean function rather than a binary relation over `Model` and `Subject`, for the same reasons as above. In UML notation, the framework is depicted in Figure 2.

Examples. We iterate the examples of Section 2, extending them to the language level.

1. Programming language types are members of a language for typing. The type checking algorithm is actually defined on the level of this language: in terms of Definition 3.1, it is the correctness function.
2. Similarly, database schemes are members of a database scheme language, which determines the semantics of the scheme; this is the correctness function.
3. Both XML DTD's and XML schemas are members of a language; again, their semantics is defined on the level of the language rather than on the level of each individual DTD or schema.
4. Programming language grammars are usually written in (some variant of) BNF. A parser for the language can then be obtained as the result of a parser generator, which embodies the semantics of BNF.
5. The possible executions of a program are never defined for individual programs, but are derived from the behavioral semantics of the programming language. Note that the semantics is hardly ever formalized; moreover, in contrast to the syntax, which as recalled above is determined by a language one level higher still, the semantics is usually determined for each programming language individually. However, in process algebra there is a well-investigated field of *Structural Operational Semantics* (see, e.g., [9,1]) which actually studies a *language* for semantics which is on the level of BNF; and this has also been applied to functional programming languages [13].

6. It is not on the level of class diagrams that it is determined what object diagrams are valid members. Rather, this is defined for all class diagrams collectively; that definition belongs on the level of the UML meta-model, alongside the membership function for class diagrams.
7. Similarly, whether or not a given program state obeys a particular class diagram is determined on the level of the UML meta-model. Thus, the UML meta-model can serve as a language for object diagrams or for program states; the correctness functions are different in each case, even though the membership function is the same.
8. It is the credo in MDA that the MOF is the meta-model for the UML meta-model. However, the different membership functions for the class, sequence and state diagrams etc., discussed in the example section of Section 2, have not been formally defined on the level of the MOF. In fact, the the technical machinery to formulate such definitions is absent. Thus, we argue that the MOF is not a language for UML diagrams in the sense of Definition 3.1.
9. On the other hand, the MOF can be seen as a language *for meta-models*, albeit in a rather trivial and uninteresting sense. Above we have recalled that the MOF is a member of itself. If we now distinguish MOF as a language from MOF as a model, and define the language MOF to have the model MOF as its only member, then the membership of that member is fixed and may as well have been defined on the language level. However, if we take into consideration that model *changes* were the reason for introducing the concept of languages in the first place, it is clear that a language with only a single model is not a worthwhile concept.
10. Scientific models of a real-world systems are usually formulated as sets of rules or equations in mathematics. It is the standard mathematical interpretation that determines the predictions the model makes; this, then, is the required correctness function for the language of mathematics.

4 Further meta-levels

The principle we have used to lift models to languages can of course be applied again to languages. That is, we might introduce, say, *theories* that are models of languages with not only a correctness function determining the membership function of the languages, but also a *semantics* function determining the correctness function of the languages. However, we currently do not see the added value of such a further layer. We therefore leave off at three.

A natural question is how the three layers of subjects–models–languages relate to the four layers of meta-modeling recognized in the MDA approach. To answer this we reiterate that in our interpretation, subjects models and languages are *roles*, or, in other words, *relative* layers: something that is a subject in one case may play the role of a model or even of a language in other situations. Indeed, in the examples above the UML meta-model has appeared in all three roles.

To illustrate this point, we take Example 4 of Section 3 above. It is well known that a grammar for EBNF can itself can be specified in EBNF format;

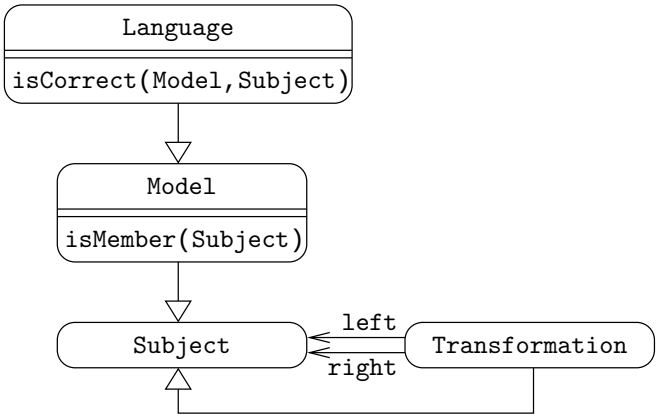

Fig. 3: UML diagram for the concepts in Definition 5.1

in the usual way, this gives rise to a parser for EBNF. Thus, EBNF occurs here both at the model and at the language level; in MDA terminology, it appears to be a reflective meta-model. However, this appearance is deceptive, for in this scenario EBNF plays two distinct roles. The EBNF parser that comes out of the parser generator implements the membership test for EBNF-as-a-model, but not the correctness test for EBNF-as-a-language; in other words, is not a parser generator. (In fact, EBNF does not have any facilities to describe semantics, so it is not possible to generate parsers which do not only parse but also interpret terms.)

5 Transformations

We now introduce, in the same framework, the concept of *transformations*. This concept is orthogonal to the subjects-models-languages hierarchy: one may transform subjects of any kinds. This orthogonality has the consequence that one may also consider, without risk of confusion, models and languages for transformations.

Definition 5.1 (transformations). *Given a universe* `Subject`*, a* transformation *is a subject* $t \in$ `Subject` *with two associated subjects* $\texttt{left}_t, \texttt{right}_t \in$ `Subject`.

The intuition is that the transformation t embodies a change which has turned $\texttt{left}_t$ into $\texttt{right}_t$. However, we make no further assumptions about the nature of this change; nor do we specify that $\texttt{right}_t$ can be derived mechanically from $\texttt{left}_t$. Typically, moreover, t may contain more information than just the pair $(\texttt{left}_t, \texttt{right}_t)$; for instance, it may contain a transformation rule that has triggered the change plus the particular way that rule has been applied in t.

Transformation models and languages. Since transformations are themselves subjects, the concepts that we have developed before apply: in particular, we

can distinguish transformation models and transformation languages. Indeed, transformations are hardly ever considered on an individual basis; instead, once a process is recognized and identified as a transformation, the question is automatically what the guiding principles for that transformation are, and how those guiding principles can be written down; these are no more and no less than the questions for transformation models and a transformation language.

As an example, we take a transformation from one data value into another, namely from 1 into 2. (Thus, this transformation t has $\texttt{left}_t = 1$ and $\texttt{right}_t = 2$.) This is an instance of many different transformation *models*, two of which are: doubling the value, and incrementing the value by one. Both these models do not just take 1 to 2 but take any natural number to a natural number; in other words, they are functions over the natural numbers. Again, there are many transformation *languages* in which such functions can be written; for instance, the language of mathematics (in which the functions appear as "$x \mapsto 2x$" and "$x \mapsto x{+}1$") or a functional language (in which the functions appear as "$\lambda x.2{\times}x$" and "$\lambda x.x + 1$").

Note that neither a transformation model nor a transformation language are themselves necessarily transformations — we do not need to specify a `left` and `right` for them.

Model transformations. An interesting issue arises when we transform models; that is, when we have a transformation t with $\texttt{left}_t, \texttt{right}_t \in \texttt{Model}$. t itself does not specify a relation between the members of $\texttt{left}_t$ and $\texttt{right}_t$. One could imagine, in addition to t, transformations of all members of the left hand side of t to members of the right hand side of t. Formally, this would come down to a family of transformations $(t_s)_s$ for all $s \in \texttt{Subject}$ with $\texttt{isMember}_{\texttt{left}_t}(s)$, such that $\texttt{left}_{t_s} = s$ and $\texttt{isMember}_{\texttt{right}_t}(\texttt{right}_{t_s})$. We again investigate some of the examples discussed before in Sections 2 and 3.

If we have such a family of transformations $(t_s)_s$ then the original model transformation t can easily be seen as a model for that family. The membership function of t is defined by: $\texttt{isMember}_t(t')$ if and only if $t' = t_s$ for some s. This means that t is both a model transformation and a transformation model.

1. An example type transformation is from $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$ (the type of functions from pairs of natural numbers to single natural numbers; for example, the addition function) to $T \to U \to V$ (the type of functions from $\mathbb{N}$, the set of natural numbers, to functions from $\mathbb{N}$ to $\mathbb{N}$; for instance the function that, given a natural number, returns a function that increments its parameter by that number), or vice versa. For the forward direction, a possible family of instance level transformations is from any function f, taken as a term in lambda calculus with type judgment $\Gamma \vdash f : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$, to the term $\lambda x.\lambda y.f(x, y)$. For the backward direction, a possible family of transformations is from any f with $\Gamma \vdash f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ to $\lambda x.f(\pi_1 x)(\pi_2 x)$.
2. There are usually many ways to store a given set of data, giving rise to different database schemes. Example transformations of database schemes arise for instance out of the normalization steps in relational databases (see,

e.g., [10]). There is always an unambiguous underlying transformation for the databases that are members of the scheme.

Other kinds of model transformations, however, are being proposed and discussed without explicit regard for the corresponding subject transformations. In other words, not all model transformations are transformation models.

4. The theory of *refactoring* (see [11,15]) is concerned with the transformation of programs into other programs. The intention is usually that the result of a refactoring has the same functionality as the original program. (Note that here we use "a refactoring" to mean "a transformation from a concrete program to a concrete program" and not for a *set* of such transformations.) Regarding a program as a model for its executions, as we did above, this means that there exists the idea that executions of the original program are transformed into executions of the refactored program. This subject-level transformation, however, is quite difficult to make precise, and it is hardly ever described on more than a verbal level.

Model transformation models In the description above, we have been careful in stressing that we were discussing transformations of particular, concrete models. However, each of these examples actually describes general principles that can be applied to many different models.

1. The type transformation mentioned above is an instance of the transformation from $(T \times U) \to V$ to $T \to U \to V$ (where T, U and V stand for arbitrary types), instantiated to the case where $T = U = V = \mathbb{N}$.
2. The relational database normalization steps are formulated in such a way that they are applicable to many different database schemes — in fact, that is precisely their point.
4. Refactorings, too, are always based on general principles. For instance, a refactoring like "encapsulate field" can be applied to any field of any class; each application is a concrete refactoring in the sense discussed above.

In those cases where they are made explicit, the "correctness criteria" (i.e., the underlying subject-level transformations) of model transformations actually always apply to the transformation *models*, and not the concrete transformations. For instance, the correctness of the transformation from $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$ to $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ was provided in the form of the subject-level transformations from f to $\lambda x.\lambda y.f(x, y)$; but this actually works on the level of the transformation model $(T \times U) \to V$ to $T \to U \to V$.

Model transformation languages Carrying the concepts one step further, we can see that the transformation models discussed above are themselves described in a particular format. In many cases, the core of the transformation model is specified by a rule consisting of a pair of terms with common meta-variables; after filling in the meta-variables, these terms are instance of a language for the models being transformed — in the terminology of this paper, the terms

are models. Applying a rule of this kind involves a form of pattern matching: whenever a model that we are interested in transforming can be obtained by a particular instantiation of (the meta-variables in) the left hand side of a rule, then the rule applies and the corresponding instantiation of the right hand side is the right hand side of the resulting transformation.

For instance, given a language for types that includes the constructor $\rightarrow$ for function abstraction, the pair $((T \times U) \rightarrow V, T \rightarrow U \rightarrow V)$, where T, U and V are meta-variables, is a term of the model transformation language defined in this way.

This, then, comprises a more or less general method whereby we can generate a model transformation language from the corresponding (model) languages. (It is *more or less* general because it hinges upon the assumption that we can use meta-variables in the way described.) Doubtless, other, also more or less general, methods can be devised, for instance using imperative principles; for instance, this same type transformation might be obtained by specifying

```
if isFuncType(T)
  let U=T.resultType;
  if isFuncType(U)
    return newFuncType (newTupType(T.parType,U.parType), T.resultType);
```

Language transformations Above we have argued that a model transformation t should itself be a transformation model for the members of $\texttt{left}_t$ and $\texttt{right}_t$. An analogous argument can be made in the case of language transformations: in this case there should be underlying model-level transformations.

We give one example, based on the general method to define model transformation languages, outlined above. This method takes model languages L_1, L_2 to arrive at a model transformation language L whose members are essentially pairs (M_1, M_2), where M_i is a pseudo-member of L_i with meta-variables thrown in, the instantiation of which gives rise to real members of L_i. Each such pair $M = (M_1, M_2)$ can in fact be seen as a language transformation, with $\texttt{left}_M = L_1$ and $\texttt{right}_M = L_2$. The underlying model-level transformations are the instantiations of M.

6 Conclusions

As announced in the introduction, this paper presents a preliminary state of ideas. Since the intention was mainly to provide a common grounds for discussion, the success of this work will have to be judged by how well the terminology presented here does at clarifying the core concepts.

Meta-models. We have gone to lengths to avoid the term 'meta-model,' even though what we have called a 'language' is very close or even coincides with others' use of 'meta-model'. Our choice for an alternative is motivated by two observations:

- The term 'meta-model', however, evokes strong associations with the MDA terminology, in particular, the hierarchy of four, absolutely numbers, meta-layers. As we have argued, we are in favor of regarding models and languages as relative rather than absolute.
- The term 'meta-model' has multiple interpretations, which we believe to be irreconcilable. One interpretation more or less corresponds to our use of the term 'language' (but for the distinction between absolute positions and relative roles). Another interpretation, however, is to take the multi-level hierarchy itself as the subject under study.

Related work. There has actually been a lot of work in clearing up the MDA terminology. For instance, [5,6,3,18] address and categorize various concepts of instantiation, and [7,8,12,17,14] present general frameworks for transformation. The main difference between this paper and the cited works is that we are interested not so much in an all-encompassing set of definitions or a classification of approaches, but rather in minimal criteria that capture the core concepts of instantiation and transformation. The result is a framework in which one can point out and discuss subtle distinctions between, for instance, model transformations, transformation models and the like.

References

1. L. Aceto, B. Bloom, and F. W. Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111(1):1–52, May 1994. LICS '92 Special Issue.
2. J. Álvarez, A. Evans, and P. Sammut. Mapping between levels in the metamodel architecture. In M. Gogolla and C. Kobryn, editors, *UML 2001*, volume 2185 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
3. C. Atkinson and T. Kühne. The essence of multilevel metamodeling. In M. Gogolla and C. Kobryn, editors, *UML 2001*, volume 2185 of *Lecture Notes in Computer Science*, pages 19–33. Springer-Verlag, 2001.
4. J. Bézivin and R. Heckel (organisers). Language engineering for model-driven software development, Mar. 2004. Dagsuthl Seminar No. 04101; see `http://www.dagstuhl.de/04101`.
5. J. Bézivin and R. Lemesle. Ontology-based layered semantics for precise OA&D modelling. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology; ECOOP '97 Workshop Reader*, volume 1357 of *Lecture Notes in Computer Science*, pages 151–154. Springer-Verlag, 1998.
6. J. Bézivin and R. Lemesle. Towards a true reflective modeling scheme. In W. Cazzola et al., editor, *Reflection and Software Engineering*, volume 1862 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag, 2000.
7. S. Bowers and L. Delcambre. On modeling conformance for flexible transformation over data models. In B. Omelayenko and M. C. A. Klein, editors, *Knowledge Transformation for the Semantic Web*, volume 95 of *Frontiers in Artificial Intelligence and Applications*, pages 34–48. IOS Press, 2003.
8. K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2002.

9. R. De Simone. Higher-level synchronising devices in Meije-SCCS. *Theoretical Comput. Sci.*, 37:245–267, 1985.
10. C. C. Fleming and B. von Halle. *Handbook of Relational Database Design*. Addison-Wesley, 1989.
11. M. Fowle, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 1999.
12. A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In *Proc. 1st International Conference on Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105. Springer-Verlag, 2002.
13. A. D. Gordon. Bisimilarity as a theory of functional programming. *Theoretical Comput. Sci.*, 228(1–2):5–47, 1999. Conference version in MFPS 1995, ENTCS 1.
14. I. Kurtev and K. van den Berg. Unifying approach for model transformations in the mof metamodeling architecture. In M. van Sinderen et al., editor, *Model-Driven Architecture with Emphasis on Industrial Applications*, volume TR-CTIT-04-12 of *CTIT Technical Report*. University of Twente, 2004.
15. T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2), Feb. 2004.
16. OMG. MDA guide version 1.0.1, June 2003. See `http://www.omg.org/mda`.
17. S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.
18. D. Varró and A. Pataricza. Metamodeling mathematics: A precise and visual framework for describing semantic domains of UML models. In H.-J. Jézéquel, H. Hussmann, and S. Cook, editors, *Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications*, volume 2460 of *LNCS*, pages 18–33. Springer-Verlag, 2002.