A Definition of “Abstraction Level” for Metamodels

Martin Monperrus
TU Darmstadt
Germany

Antoine Beugnard
TELECOM Bretagne
France

Joël Champeau
ENSIETA
France

Abstract

In model-driven software development, the first-class data are models, which are all structured by a metamodel. In this paper, we propose a definition of abstraction levels for metamodels based on set theory and compatible with MOF. We claim that splitting metamodels into different abstractions levels raise their organizational quality. We present application cases of this statement.

1. Introduction

Model-driven development is an approach to software development that is supposed *to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively* [1]. Practically, models can ground code generation and model transformation.

Every model is structured by a metamodel, and is said to conform to it [2]. Hence, metamodels are very important artifacts of a model-driven development. They can be composed of several hundred of classes, such as in the UML metamodel [3]. Dealing with big metamodels involves organizing them with sub-packages. The problem is that packages are created without any constrains, since the package semantics is very weak. Packages only denote an organizational point of view.

In this paper, we claim that metamodels can be semantically organized following their internal abstraction.

Hence, our contribution is a definition of what is an *abstraction level* in a metamodel. This definition is made with the set theory. On top of this well-formed definition, we explore three application cases. A significant benefit of our definition is the ability to remove the redundant definition of the highest abstraction level in multiple metamodels (e.g., UML, ECORE and AADL). This paves the way to better interoperability between these metamodels and the corresponding tools.

The remainder of this paper is organized as follows. In section 2, we give a definition of *abstraction level* for metamodels. Then, we discuss in section 3 how this definition can be applied. We finally explore related works and conclude.

2. A New Definition of Abstraction Level

Let’s consider a metamodeling architecture with the concepts (or corresponding concepts) of class, inheritance, association and association specialization. For example, MOF [4] satisfies these conditions.

These are mapped onto four sets:

- S_C the set of classes.
- S_I the set of inheritance relationships, a set of ordered 2-tuples of elements of S_C. We have $S_I \subset (S_C, S_C)$.
- S_A the set of association relationships, a set of ordered 2-tuples of elements of S_C. We have $S_A \subset (S_C, S_C)$.
- S_S the set of association specialization relationships, a set of ordered 2-tuples of elements of S_A. We have $S_S \subset (S_A, S_A)$ i.e $S_S \subset ((S_C, S_C), (S_C, S_C))$.

Definition 1: An abstraction level $\mathcal{L}$ in a metamodel is a subset of classes such that every relationship which cross the frontier have the same orientation and are only inheritance and specialization.

Let $\mathcal{L} = (L_C, L_A)$ such as $L_C \subset S_C$ and $L_A \subset S_A$. Let $\overline{L_C} = S_C \setminus L_C$ and $\overline{L_A} = S_A \setminus L_A$. $\mathcal{L}$ is an abstraction level if and only if : $\nexists (x, y), (z, t) \in S_I, x, t \in L_C \wedge y, z \in \overline{L_C}$ and $\nexists (x, y), (z, t) \in S_S, x, t \in L_A \wedge y, z \in \overline{L_A}$.

In what follows, we will use the expression *upper levels* to indicate more abstract levels. However, these upper levels are assigned smaller numbers in order to get a top-down representation and unbounded positive numbering It is the opposite of the OMG numbering strategy.

In figure 1, a metamodel of an aircraft company is presented. While it can simply be considered as a model, in a pure MDE viewpoint this is also a metamodel (cf. [5]). This metamodel is extremely simplified in order to maintain its legibility. The metamodel uses the following modeling features:

- inheritance (exemplified as (1) *LongRangeAirliner inherits Vehicle*)
- specialization of associations (exemplified as (2) *CFM56C propulses A340*)

According to our definition of ”abstraction level”, there are 3 abstraction levels in the metamodel of figure 1. They are showed figure 2. The uppermost one is very abstract and deals with abstract concepts such as *Vehicle* and *Engine*. The middle abstraction level is about *LongRangeAirliner*. The lowest abstraction level is dedicated to the A340.

Accepted for publication in the 7th IEEE Workshop on Model-Based Development for Computer Based Systems (MBD'2009) co-located with ECBS'2009

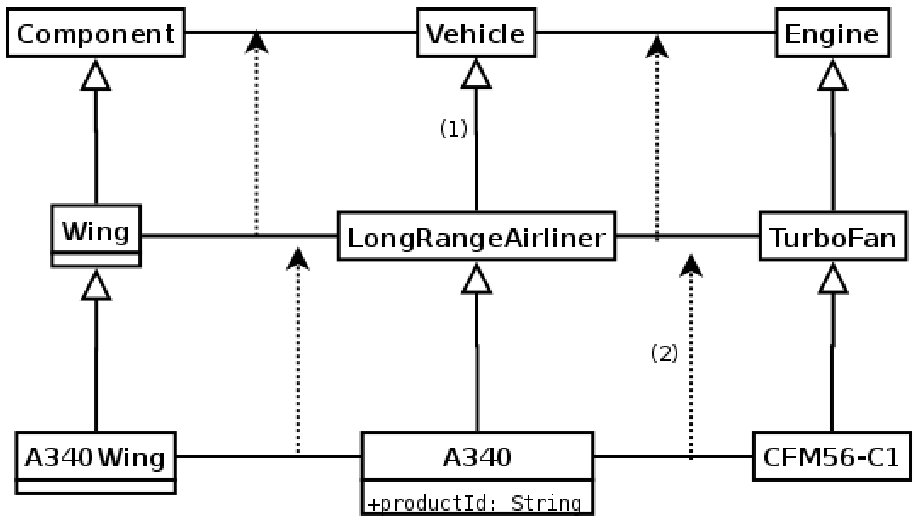

Figure 1. A metamodel of an aircraft company

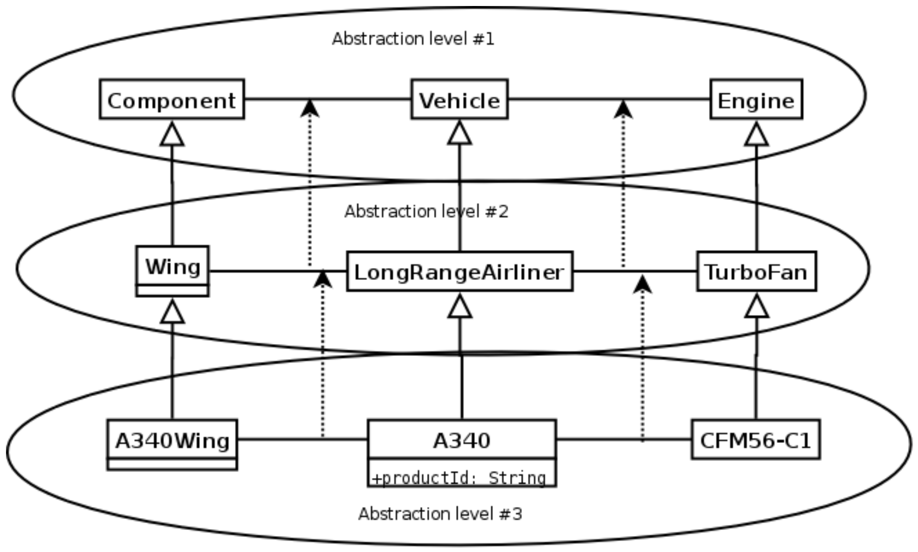

Figure 2. The metamodel contains 3 abstraction levels

3. Application cases

In this section, we explore the use of abstraction levels in different application cases.

3.1. Splitting existing metamodels into abstraction levels

Since one can assume that metamodelers have already implicitly used abstraction levels in their metamodels, we explored whether abstraction levels may already exist in real-world metamodels[1].

1. The exploration was semi-manual. A fully automatic search of abstraction levels is an instance of (k,2)-partite graph problem (cf. [6]).

Let's consider the *Ecore* metamodel [7] shown in figure 3, put at the end of the paper for reasons of space. Since the *Ecore* metamodel was created by people who are truly aware of the essence of metamodeling, it is likely to contain or almost contain abstraction levels. Indeed, this metamodel contains 3 abstraction levels.

The figure 4 shows the highest abstraction level of the *Ecore* metamodel. This abstraction level is highly reusable. It defines a root class *EObject*, the notion of a named element *ENamedElement* and the notion of annotation *EAnnotation*. Annotations are like comments in programming languages, they are simply essential for maintenance and reuse. The *Ecore* version of annotations is powerful, it's a reference to an *EObject* with a double mapping $String(source), String(key) \rightarrow String(value)$. It is a

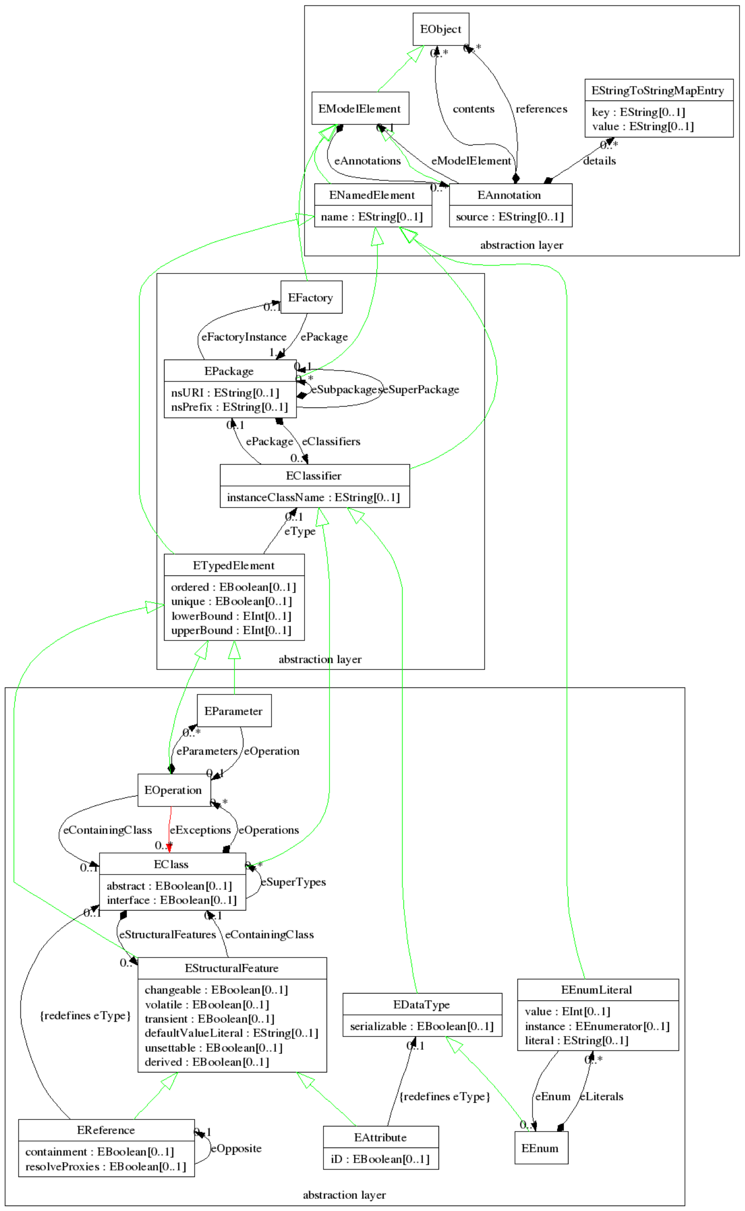

Figure 3. The *Ecore* metamodel split into three abstraction levels

mean of embedding semantic information in annotations, *à la* Javadoc. More or less the same abstraction level is found in a lot of other metamodels, such as UML [3], AADL [8] and others. This is a duplication which could be avoided with the use of abstraction levels. Hence, if it has been explicitly an abstraction level, a documentation generator would work with *Ecore*, UML and AADL.

The two other abstraction levels of figure 3 are well-formed abstraction levels just by changing the *eExceptions* reference from EClassifier to EClass This change makes appear a new layer which is really meaningful. It forbids to be able to throw an Eclassifier, i.e. potentially a EDataType or an EEnum. In regard to existing exception mechanisms, it is better to only throw Eclass. Indeed, the EMF editor hard-codes this rule whereas it could be made explicit directly into the *Ecore* metamodel.

Splitting a single-level metamodel into smaller leveled metamodels is one way of addressing reusability. Actually, by construction, a more abstract level is totally independent from more concrete levels, i.e. there are no cross-dependencies between two adjacent levels. All relationships (inheritance, specialization) go in one and only one direction, from L_{n+1} to L_n. Each level does not depend on the levels below it. Thus, upper levels become totally reusable parts

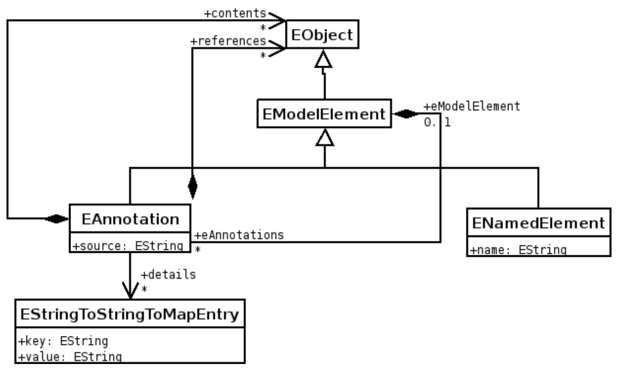

Figure 4. A reusable part of the *Ecore* metamodel

(see figures 2,3). This provides a more flexible and extensible metamodeling framework in which each abstraction level can be refined. The upper level of the figure 2 can be used to define another kind of vehicle like an automobile and the second abstraction level of this figure can provide the support for a new type of *LongRangeAirliner*.

3.2. A measure of abstraction

Since metamodels are important artifacts of Model Driven Engineering, we need to measure them in order to characterize them, as well as extracting a typology, patterns, etc. Indeed, quality assessment and assurance is not possible without a complete picture of the characteristics of a software product [9]. The definition of abstraction levels we propose naturally leads to measuring the abstraction of a given metamodel.

Definition 2: The quantity of abstraction of a given metamodel is the maximum number of stacked abstraction levels.

Every measure has a scale, which can be nominal, ordinal, interval, ratio or absolute (see [10], [9] for more details). The measure of abstraction presented in this paper represents a count (the number of stacked abstraction levels). Hence it has an absolute scale. This scale permits a full range of descriptive statistics to be applied. This measure of abstraction defines a relation R which is *has more or an equal number of abstraction levels*. This relation defines a mathematical order relation between metamodels.

Weyuker's set of nine properties [11] provides a framework to evaluate a measure. These abstract properties help to characterize measures and to provide correct definition for them. For instance, the first axiom reflects the intuition that a measure should give different values for different programs. Among these nine axioms, those numbered 5, 6 and 9 are link to monotonicity and increasing of complexity when programs are composed. Since no such composition relation between metamodels is considered, these properties do not make sense. When considering the other six properties, the measure of abstraction fulfills all of them.

3.3. A criterion for driving the metamodeling process

Once we are dealing with many levels of modeling, the relationship between two levels must be clearly identified and defined. In object-oriented modeling, there are abstractions on the program side and concrete data on the instances/memory side. When metamodeling this clear separation is fuzzier. Our definition of abstraction level help to clearly identify when and how to introduce a new abstraction level.

In this paper, we call *abstraction gap* the need for splitting a metamodel into two smaller metamodels, so that the only relationships crossing the abstraction frontier are those of *inheritance* and *specialization*. Every level corresponds to a given abstraction level. The example above illustrates this point. Even though the *Vehicle* and *LongRangeAirliner* entities are both abstractions, they are not at the same level of abstraction.

The abstraction gap can be jumped over in two directions, identifying either a more abstract level or a more concrete level. The first is called a *bottom-up jump* and the latter a *top-down jump*.

A top-down jump involves creating a new level. This new level will contain less abstract concepts. Making a top-down jump (creating a new lower level) is triggered by the need to specialize an association as shown in figure 5. Top-down jumps are usually done in a refinement process.

A bottom-up jump is also the creation of a new level, which will contain more abstract concepts. A bottom-up jump (creating a new upper level) is triggered by the need to generalize an association, as seen in the figure 6.

Depending on the jump type, a new modeling level will be added above or under the level currently being manipulated. In the case of a bottom-up jump, the numbering of the level being handled will be changed. As mentioned above, a top-down jump is a kind of refinement, which is a classic way of modeling and creating a system. However, the need to explicit concrete cases in order to extract abstractions is a usual cognitive process. In fact, this abstraction process is often done internally, by the brain, without the help of any notation or tool. For abstractions which are hard to detect, concrete cases must be described before obtaining the common factors, i.e. the abstraction itself.

What triggers the creation of a new abstraction level? We saw in the previous example that creating a sub-class does not necessarily trigger a level jump. Actually, classifying (sub-classing or generalizing) is a way of creating abstraction levels. However, it is not suitable to create a new abstraction level each time a classifier is created. The reason is mainly because models would become too much clustered, i.e the modeling would not be done with the right grain. This does not mean that it should never be made along with a jump over the abstraction gap; that decision is up

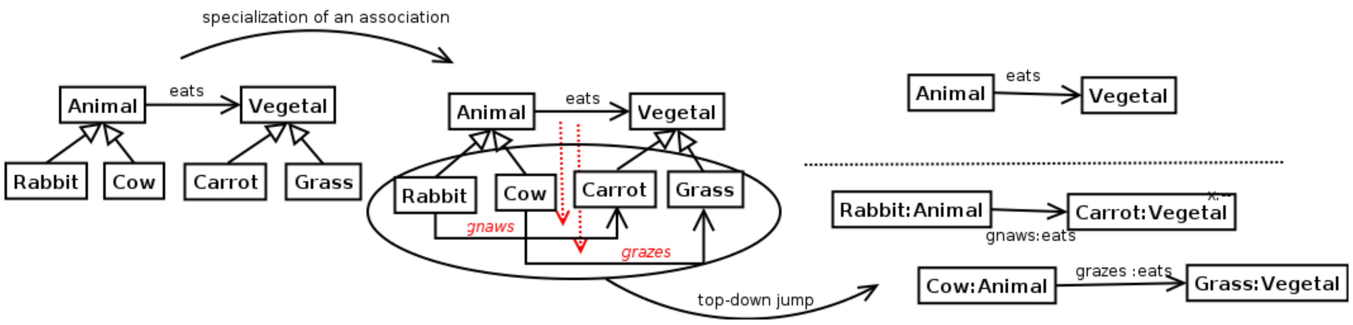

Figure 5. Top-down jump over the abstraction gap

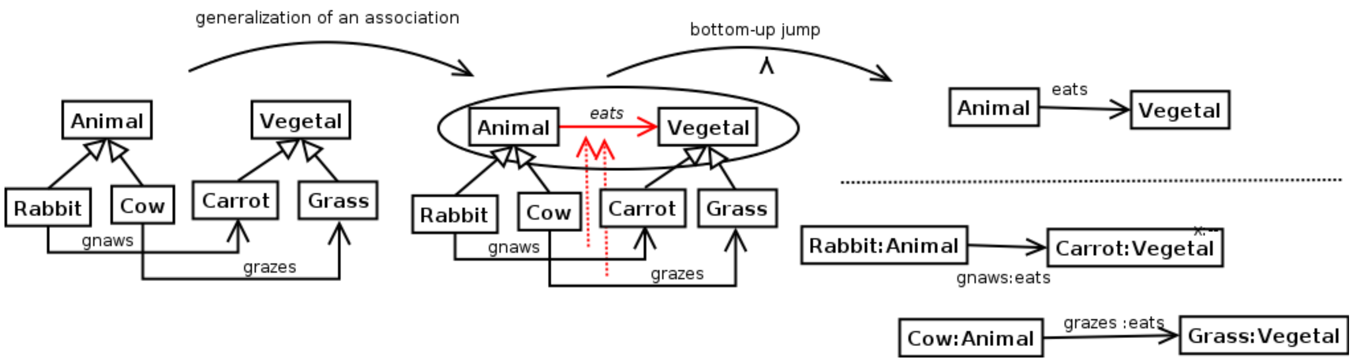

Figure 6. Bottom-up jump over the abstraction gap

to the modeler. With his or her domain knowledge, (s)he is able to determine the correct abstraction grain, i.e. where the addition of a classifier will act as a jump over the abstraction gap.

Specializing associations is much more different. Unlike sub-classing, specialization of associations restricts the possible models. Another specialization is to change the cardinality of an association. Specializing the associations or modifying their properties (e.g. multiplicities) represents a refinement and a restriction of possible models. That is why association specialization is a key potential trigger. The need to specialize an association is a serious indication of a modeling level change, i.e. a trigger to create a new abstraction level (for both bottom-up and top-down jumps).

4. Related work

In 1979, Parnas stated [12] he had not found *a relation "more abstract than" that would allow me to define an abstraction hierarchy*. Our definition allows to define a *more abstract than* relation between class modules (i.e.; packages). Note that the stack of abstraction levels exposed in this paper follows the module usage hierarchy pattern defined by Parnas.

Atkinson et al. explored [13] issues and solution to multilevel modeling. As Atkinson et al. we believe that multilevel modeling is useful. Compared to their work, our contribution is fully operationalized in the MOF world, so as to give a clear and ready-to-use way of manipulating abstraction levels to MOF users.

In [5], Kühne aims to start establishing a consensus on the notions of model and metamodel in model-driven engineering. While the multilevel issue is not central to this paper, it appears in background. Our definition of an abstraction level is equivalent to the ontological model-of relation of Kühne. Our contribution is a a concrete definition of the ontological model-of relation of Kühne.

In [14], Sangal et al. manage the dependencies of software through a matrix. This dependency matrix ground the expression of explicit design rules. Our contribution refines their work by defining a special kind of dependency for metamodels based on abstraction.

In the UML2 specification [3], the issue of abstraction level is partially addressed by the Abstraction concept. Our work clarifies the semantic of the abstraction dependency. In a sense, our work defines a clear way to get a *possibly infinite number of meta-layers* (UML 2.0 infrastructure specification, § 7.2.7, p. 28).

Furthermore, the profile mechanism of UML [3] gives the user a way to explicitly handle 2 levels of abstraction: the classes in M_1 and the meta-classes of the profile in M_2. The existence of this mechanism in such a standard demonstrates the need to manipulate several levels of abstraction. However, this mechanism is not sufficient. Our definition is a objective criterion that allows to define more than two abstraction levels.

5. Conclusion

Model-driven system development involves many points of views. These points of views may have a different analysis grain, which means considering different abstraction levels. Metamodeling is a powerful modeling technique, in which it is possible to deal with different abstraction levels. However, their was no definition of what is an abstraction level in a metamodel.

In this paper, we have proposed a well-defined and unambiguous definition of abstraction level in metamodels. This contribution paves the way to better handling of abstractions in metamodels and during the metamodeling process. Several applications of the abstraction level definition have been presented. We defined a measure of abstraction for existing metamodels. We showed how to improve existing metamodels, by isolating reusable abstraction levels. Finally, this definition of abstraction in metamodels gave insights about how to handle abstraction while metamodeling. Future work will explore how to leverage the modularity enforced by abstraction levels to build reusable model transformations.

Acknowledgement

This work was partly supported by the *feasiPLe* project, Federal Ministry of Education and Research (BMBF), Germany.

References

[1] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, pp. 25–31, February 2006.

[2] J. Bézivin, "On the unification power of models," *Software and System Modeling*, vol. 4, pp. 171–188, May 2005.

[3] OMG, "UML 2.0 superstructure," tech. rep., Object Management Group, 2004.

[4] OMG, "MOF 2.0 specification," tech. rep., Object Management Group, 2004.

[5] T. Kuehne, "Matters of (meta-) modeling," *Software and System Modeling*, vol. 5, no. 4, pp. 369–385, 2006.

[6] H. A. Muller and J. S. Uhl, "Composing subsystem structures using (k,2)-partite graphs," in *Proceedings of the IEEE Conference on Software Maintenance*, 1990.

[7] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose, *Eclipse Modeling Framework*. Addison-Wesley, 2004.

[8] SAE, "AADL Standard," tech. rep., Society of Automotive Engineers, 2006.

[9] B. Henderson-Sellers, *Object-Oriented Metrics, measures of complexity*. Prentice Hall, 1996.

[10] H. Zuse, *Software Complexity*. Berlin: Walter de Gruyter, 1991.

[11] E. J. Weyuker, "Evaluating software complexity metrics," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1357–1365, 1988.

[12] D. L. Parnas, "Designing software for ease of expansion and contraction," *IEEE Transactions on Software Engineering*, vol. 5, pp. 128–138, Mar. 1979.

[13] C. Atkinson and T. Kühne, "The essence of multilevel metamodeling," in *Proceedings of the 4th International Conference UML'2001* (M. Gogolla and C. Kobryn, eds.), vol. 2185 of *LNCS*, pp. 19–33, Springer, 2001.

[14] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," *SIGPLAN Not.*, vol. 40, no. 10, pp. 167–176, 2005.