Towards an Open Meta Modeling Environment

Bernhard Volz, Stefan Jablonski

University of Bayreuth
Universitaetsstrasse 30
95447 Bayreuth, Germany

{bernhard.volz, stefan.jablonski} @ uni-bayreuth.de

ABSTRACT

Conventional modeling environments support either only a two layered meta hierarchy or do not provide (full) support for advanced modeling paradigms that go beyond the capabilities of the Meta Object Facility (MOF). Within this article we introduce the foundation of a meta modeling environment that supports Powertypes, Clabjects, Deep Instantiation and Materialization.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – *frameworks, patterns*

General Terms

Algorithms, Management, Design, Standardization, Languages.

Keywords

Meta Modeling, Meta Hierarchies, Linguistic Meta Model, Powertypes, Clabjects, Deep Instantiation, Materialization

1. INTRODUCTION

The ability to extend and adapt modeling languages is a key requirement for domain-specific modeling. Modeling languages and their constructs must frequently be adjusted to changing requirements. Therefore tools supporting this evolutionary process are crucial for applying the domain-specific modeling / language paradigm in practice.

A well-known approach to introduce domain-specific modeling constructs is the provision of profiles that alter contents of meta models; one incarnation of this approach manifests in the profile mechanism for the Unified Modeling Language (UML) [1, 2]. Despite that UML profiles are very widespread, their use and application is limited by means of the UML specification in such a way that the original semantics of UML constructs cannot be changed or even removed. Only conceptual additions are possible that make semantics more precise or add semantics in those places where it was left open (intentionally) by the initial specification. Thus, a list of scenarios can be named in which the provision of a UML profile is feasible and in which standard UML tools can be used further. However, modeling languages whose constructs cover different semantics than those of the UML are not among these scenarios. Instead, users are forced to design such a language from scratch and cannot make use of standardized UML tools. Especially in combination with more advanced modeling patterns such as (Extended) Powertypes [3, 4], Clabjects [5], Materialization [6] or Deep Instantiation [7], the semantics of UML constructs is easily violated and model architects are forced to implement their own domain-specific language along with a set of accompanying tools.

A common way of how a new (domain-specific) modeling language can be created is to define the abstract and concrete syntax of a language along with a corresponding model [8] using specialized frameworks such as Xtext [9] for the Eclipse platform or the Microsoft Domain-Specific Language tools for Visual Studio .NET [10]. These and other approaches which apply the same language development paradigm are restricted to only two layers within a meta hierarchy and therefore do not provide the same flexibility as those tools which cover more than two layers (e.g., ConceptBase [11], ATOM3 [12], GME [13]). Using more than two layers within a meta hierarchy is helpful because transformations and/or links between different language (versions) become easier when languages are defined on a common basis. Thus, providing such a third layer ensures interoperability and supports the evolution of languages.

All approaches mentioned above – independent from supporting two or three layers – do not support afore mentioned modeling patterns (e.g., Powertypes, Clabjects). One reason is that usually the Meta Object Facility (MOF) [14] or derived standards (e.g., the Ecore model of the Eclipse Modeling Framework (EMF) [15, 16]) are used as a basis for these approaches. Another reason is that features of those patterns are violating fundamental concepts of these approaches (e.g., the unification of type- and instance-facet as proposed by the Clabject modeling pattern is violating fundamental axioms of ConceptBase).

Furthermore, we fully agree with the idea of Atkinson and Kühne [17] who stipulate the separation of linguistic and ontological aspects within meta modeling. The linguistic aspect of a meta model is responsible for representing (meta) models; the ontological aspect covers the content of a meta model. As a result, models of different nature are stored using the same linguistic notions and can therefore be "understood" by all tools that support this storage language – whatever the content of a model is all about. This feature is also a main concern within the definition of UML profiles: a UML profile should never change storage structures of models [2]. Pursuing this approach means to have a tool environment which is capable of creating and maintaining arbitrary models that – even more important – use different meta models as their basis. As a consequence, the separation between linguistic and ontological aspects allows us to implement a modeling environment that supports these advanced modeling paradigms (modeling patterns) and at the same time can be used for creating any kind of domain-specific model. To our knowledge, such an environment does not exist so far. This contribution presents the design rationale and the architecture of such a modeling environment.

Inspired by the work of Atkinson and Kühne [17] we introduce a Linguistic Meta Model (LMM) that allows for representing arbitrary hierarchies of meta models. The LMM is the basis for our Open Meta Modeling Environment (OMME) which is a set of plug-ins for the Eclipse Platform and that allows for creating models with an arbitrary number of meta layers and that interprets concepts such as (Extended) Powertypes, Deep Instantiation, Materialization and Clabjects.

The remainder of this article is structured as follows: Section 2 introduces the design rationale of the LMM while Section 3 details the implementation of the LMM. Section 4 then shortly introduces the complete OMME platform together with an evaluation of our prototypical implementation. Section 5 finally concludes this paper by providing an overview on future work.

2. DESIGN RATIONALE

The following section explains the design rationale behind the development of the Linguistic Meta Model (LMM).

R.1. Support for Flexibility

The LMM is not only intended to represent models that consist of several meta levels/layers but also to facilitate flexibility in modeling by integrating advanced modeling concepts such as (extended) Powertypes, Deep Instantiation, Materialization and Clabjects. These patterns aim at reducing redundancies and increase the expressiveness of models and modeling languages. Integrating these patterns, however, means to at least partially break with conventional modeling semantics as it can be found within the concepts of generalization, specialization and instantiation. Thus, the LMM does not consist of a special modeling paradigm – it is merely used for representing model structures. Modeling rules are enforced by an additional component which is not part of the LMM.

R.2. Support for Arbitrarily Structured Meta Hierarchies

MOF [14] and derived modeling efforts such as EMF [15, 16] are founded on top of a linearly ordered hierarchy of meta levels which are related via *instanceOf* relationships. While this design is advantageous with respect to a simpler implementation of modeling environments, modeling patterns such as Powertypes cannot be implemented on top of such a strictly ordered stack since they might require additional relationships to be drawn between levels and/or their content.

The LMM in general supports any kind of relationship between modeling levels since it merely represents model structures but no semantic. In practice, the available relationship types are *instanceOf* and *references*; the latter one defines a loose relationship between two levels and merely stands for the "use" of entities located at the related level within the referencing level but does not impose a restriction on this use. Furthermore, the order of levels can be chosen freely; also non-linear layouts (cf. [18]) can be represented with the LMM.

R.3. Abdication of Self-Describing Top-Level

Within MOF [14], M3 is self-describing, meaning that M3 can be seen as an instance of a level M4 which only repeats the contents of M3 – or in other words: M3 is an instance of itself. Whilst self-description plays an important role within the theory of meta modeling (especially considering the way how the OMG defines meta modeling within MOF), it is not necessarily required in practice. Thus, within the LMM, the highest logical level is only making statements about the logical content of the next lower level. Nevertheless, a self-describing (logical) top-level can still be defined.

R.4. Use of Neutral Naming

Meta modeling and also the development of domain-specific languages are tightly bound to the realm of software engineering or – more general – computer science. Thus, terms from these realms are used for describing entities such as "class" or "object". The LMM avoids these terms and uses neutral naming – also considering that many of these do not fit because of the integration of advanced modeling patterns (cf. R.1.); here, especially the term "clabject" is hard to understand and the LMM speaks of "concepts" instead. Also, a differentiation between "properties" and "attributes" does not exist.

3. THE LINGUISTIC META MODEL

The LMM is conceptually structured into three packages, namely *References*, *Extensions* and *Core*. The *Core* package contributes basic notions and elements, the *Reference* package types for relating elements of different and the same model with each other and the *Extensions* package provides advanced modeling patterns such as (Extended) Powertypes and Deep Instantiation (cf. R.1).

Please note that we prefer to use the term "concept" instead of "class" or "object" (cf. R.4); further, within the LMM a concept represents a Clabject (cf. R.1). Concepts are used for describing entities which have an instance- and a type-facet at the same time such as the result of a Powertype instantiation (the result of a powertype instantiation is by definition of the Powertype pattern an instance of the powertype and a specialization of the partitioned type at the same time and thus has an instance and a type facet).

Basic elements within the *Core* package are MModel, MLevel, MPackage and MConcept (cf. Figure 1). MModel represents a complete ontological model that can consist of several meta levels which are given as instances of the type MLevel. The top level within a hierarchy does not have to be self descriptive but may be defined axiomatically (cf. R.3). Each level consists of at least one package, represented by an instance of type MPackage. Packages within the LMM are related to packages as known from Java [19] or namespaces as known from C# [20] and provide support for structuring models. Thus, each package contains one or more elements; in the current version, the LMM provides two types for defining elements, namely concepts and enumerations. Concepts are represented as instances of the LMM type MConcept, enumerations are omitted here for simplification reasons only.

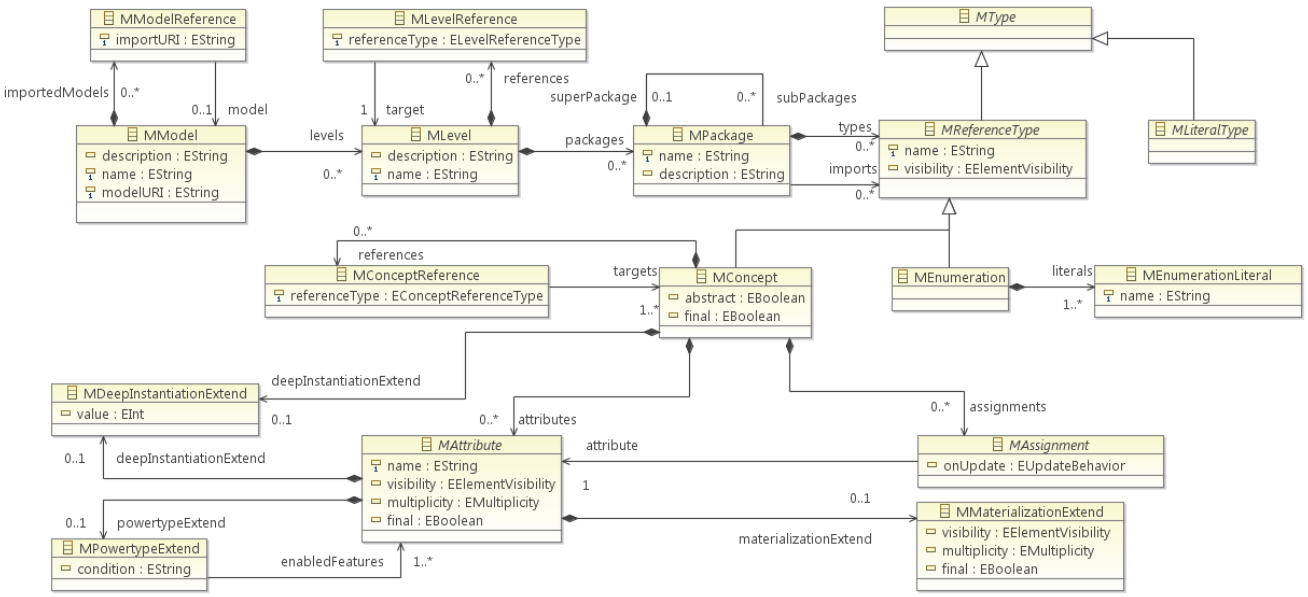

Figure 1. Essential parts of the Linguistic Meta Model; not shown are the type hierarchy along with enumerations and specializations for attribute definitions and assignments

Concepts define attributes (MAttribute) in the type-facet and declare values for attributes (MAssignment) in the instance-facet.

Within the LMM we distinguish two different kinds of element types, namely *reference types* and *literal types*. Reference types are those which can be defined by the user – concepts and enumerations. Literal types, in contrast, are pre-defined; the LMM provides types for representing numbers (MInteger, MDouble), strings (MString), boolean values (MBoolean), uuid's (MUUID) and fully qualified names (MFQN). Fully qualified names are used for referencing model elements by specifying the complete path to that element within a model whenever a strongly-typed reference should be avoided (one could compare FQNs to defining a reference to java.lang.Object within Java).

Consequently, for each data type, a corresponding attribute type and assignment type exists since then the implementation of the LMM and accompanying tools becomes easier. Currently, the LMM is implemented using the Ecore model of the Eclipse Modeling Framework [15, 16] and a textual syntax is provided using the Xtext Framework [9]. This textual syntax is specified by means of a grammar which is able to provide logic for assessing whether a given value assignment matches the definition of an attribute by its type. If different types of attributes are distinguished in the grammar and in the model, these type checks can be derived from the grammar automatically and are performed by the generated parser avoiding a "manual" implementation.

Relationships between models and model elements are described within the References package. Models may reference other models in order to use or extend their contents. Levels reference other levels in order to establish a hierarchy and concepts reference other concepts in order to describe logical relationships among them.

A reference between models does not impose any specific semantic besides a "uses" relationship. By "using" other models, elements of these models (levels, concepts and enumerations) become visible within the referencing model; e.g., a concept can be used as a type for an attribute or as a source for a specialization relationship.

Levels might also reference other levels. The LMM currently supports three types of references, which are given as members of the enumeration ELevelReferenceType. *instanceOf* describes an instantiation relationship between two levels (i.e., in a strict modeling sense all elements of the reference source are instances of elements within the target of the reference [21]) while *references* describes a more loose coupling between two levels which also allows for specifying specialization relationships across level boundaries – a feature which is needed for the implementation of Powertypes (cf. R.1). *alignedWith* merely expresses a logical alignment of levels and can be used in practice when two different models are linked together into one but the total number of levels in each model is differing and/or the structure of the level hierarchy is different. Please note that the LMM does not impose restrictions on which levels can be related and that therefore arbitrary level hierarchies can be created (cf. R.2).

Different kinds of relationships between concepts are enumerated by EConceptReferenceType. It comprises four literals, namely *instanceOf*, *extends*, *partitions* and *concreteUseOf*. *instanceOf* marks the source of a relationship between concepts as being an instance of the target while *extends* marks the source as being a specialization of the target. A relationship of type *partitions* assigns the role Powertype to the source and Partitioned Type to the target and therefore reflects the Powertype pattern (cf. R.1). Last but not least, the *concreteUseOf* relationship type is used to specify a specialization of the instance-facet of a concept.

All other elements which are needed for implementing the advanced modeling patterns (cf. R.1) are placed within the Extensions package and carry a name that ends on *...Extend*. These are counter values for deep instantiation (MDeepInstantiationExtend), a link between two or more attributes for extended powertypes (MPowertypeExtend) and

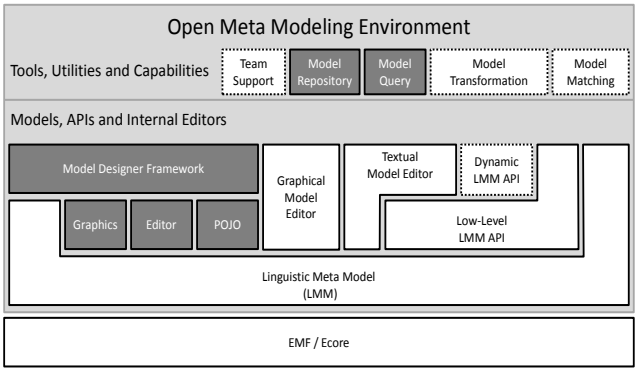

Figure 2. Layered architecture of the OMME; white boxes with straight lines are implemented, a first version of the gray boxes is available and boxes surrounded by a dashed line are future extension plans

information on how attributes are to be emulated for materialization (MMaterializationExtend).

4. OMME – ARCHITECTURE AND USE CASES / EVALUATION

The Open Meta Modeling Environment (OMME) is an implementation of the LMM together with a set of editors and frameworks that ease the development of models and programs that work with such models. It is implemented on top of the Eclipse platform leveraging on frameworks such as EMF, GMF [22], GEF [23] and Xtext [9]. Currently, we have implemented the LMM as an Ecore model, a textual and a graphical editor for creating and modifying models (with Xtext respectively GMF) and a static API that allows for managing model content programmatically. A new framework that resembles GMF but, in contrast to GMF, is able to dynamically create editors without restarting the whole development environment, is currently under development.

Further, we also provide a model repository and an SQL-like query language to query models stored within that repository. While conceptually such a repository allows for sharing models between multiple users, sophisticated support for the synchronization of models has not yet been added.

Up to now, we can create models with our stack of tools quite comfortably and our tools support model evolution by means of allowing for introducing changes to all levels within a meta hierarchy. Basic consistency checks are provided and more sophisticated ones can be added using the Check language included with Xtext. Whenever a change in a level within a hierarchy causes subsequent errors in other levels, these errors are highlighted allowing for an easy identification of where adaptations are needed because of this change.

All content within a model is still presented to the user (i.e., the person who is creating a model) expressed in terms of the LMM. As a result, especially low levels appear to be complex since terms of the LMM are used to describe contents rather than the terms defined within the (logical) meta model. For instance, within an hierarchy describing ER models a user would expect the terms “Entity” and “Relationship” for defining a schema but not “Concept”; this is even more worse when looking at instances of an ER schema – here also “Concept” is used instead of “Customer” or “Address”. Therefore we are thinking of providing a dynamic API to the LMM that allows using terms defined within a higher level for describing content of depending levels.

As a first use case, we created a model for the Perspective Oriented Process Modeling (POPM) approach, as introduced by [24]. A complete stack for this model spans three levels; the top level (M3) contains a basic notion of how a process modeling language can be expressed while the second level (M2) consists of a generalized process modeling language together with several specializations (e.g., for specifying clinical pathways). Process models are then stored on the third level (M1). In this way, we gain the following advantages which are either not possible with conventional modeling paradigms or require a heavy-weight solution.

- The content of the M3 level describes two things within the same model – first, a basic notion for processes is given, in our case these are directed graphs which may contain cycles. As a matter of fact, this part of the model defines concepts such as Node and Flow (for connecting nodes). Future constructs of a process modeling language will be derived from these two central types. Second, the M3 level contains basic restrictions for a modeling environment by providing a Powertype for each type, here for Node and Flow. The Powertype for Node, namely NodeKind, defines whether a construct of a process modeling notation such as Process can have inbound and/or outbound connections and produce/consume data.
- M2 then first defines a basic process modeling language. Each construct of this language is an instance of one of the Powertypes located at M3. Thus, a Process is an instance of NodeKind and, because of the semantic of the Powertype pattern [3, 25], at the same time a specialization of Node. Therefore, for each attribute of NodeKind, a value needs to be provided by Process; this statement declares that a Process supports inbound and outbound connections and consumes and produces data. A construct which does not support inbound connections is a Start interface which denotes the beginning of a process. Here it is clearly visible that the integration of the Powertype pattern is extremely helpful since no additional constraints need to be specified – everything is defined within the model and expressed in terms of the LMM.

  Additionally, M2 contains a process modeling language for each application domain as a specialization of the general process modeling language. Constructs within these languages either specialize constructs of the general language or are created “from scratch” by instantiating a Powertype located at M3. The provision of a domain-specific language which is based on a general language supports the exchange of process models between domains since languages are easier to relate in case they are based on the same fundamental concepts.
- Finally, M1 contains process models. Each process model consists of several steps (processes) which are interconnected and that may consume and produce data. Data items can be either described straight-forward in an informal manner or by a link into an ER model. Linking models of different kinds is easy within OMME since all models share the same representation (LMM). The use of elements of a “different” model, however, needs to be prepared – e.g., the process modeling language or even better the model on M3 can

define such a link. Since we cannot foresee which models are to be connected and since we also do not want to force users to apply specific models only, adapting higher levels of the overall model hierarchy is crucial.

Beside the text editor for the LMM, we also defined a graphical editor with the help of the OMME Model Designer Framework prototype; a screenshot with a simple process model is shown in Figure 3.

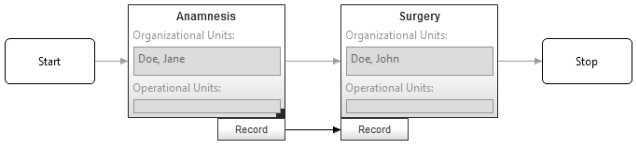

Figure 3. A sample POPM process modeled with the OMME Model Designer Framework (screenshot)

Because all models for the Model Designer Framework (cf. Figure 2) are defined upon the LMM, an adaptation always takes place within a normal text editing session, i.e., it can be performed using the standard text editor for the LMM without reloading, regenerating or even recompiling plug-ins. The Model Designer Framework requires three models as input: First, a graphical representation for processes is defined. This graphical model contains references to pre-defined figures and figure elements which can be assembled to form new shapes. Only if none of the given figures meets the users' requirements, an implementation of a figure (in Java code) needs to be provided along with a mapping, i.e., a figure in the model is linked to a Java class that is responsible for drawing this figure (in Figure 2 this is the model called POJO). Therefore, if standard figures are used, the POJO model does not need to be provided; instead, the predefined model can be used. Last but not least, a mapping between a domain model (here for POPM) and the graphical model of the Model Designer Framework needs to be provided which specifies how an element of the domain model is to be represented. Since we consider this mechanism as more or less "usual" with respect to other tool implementations, we do not want to go into more detail about this mapping. Instead, we want to shortly introduce a second use case because it also shows how advanced modeling concepts facilitate modeling.

As a second use case we chose ER modeling; one reason is that we want to use it within our process modeling approach POPM. Second, ER modeling is a good example for what Materialization [6] can be used.

Again, we are using a three-level architecture consisting of M2, M1 and M0. M2 contains the general description of an ER model, meaning it consists of entities, attributes and relationships (we want to restrict ourselves in the following to entities and attributes for lucidity; however, also relationships fit into this scenario without significant changes). An entity contains attributes and attributes have a type and a value. Thus, Entity and Attribute are defined as concepts and the composition between Entity and Attribute as an attribute attrs which is assigned to Entity (cf. Figure 4). A concrete model then specifies a Customer as an instance of Entity and the Name of this customer as instance of Attribute. Then, in the sense of the strict meta modeling approach [21], the attribute Name is assigned to Customer by placing a reference within the attribute attrs as an instance of the LMM type MAssignment.

For a normal scenario in which only schemas are modeled, this is sufficient; entities can be modeled along with their attributes. But if also instances of this schema shall be placed within the same model (i.e., that the method "ER" is applied to a concrete example – thus the model contains the method definition and the application of it [26]), two problems arise: First, an attribute needs to be declared that contains values for an attribute (the name "Otto" needs to be stored somehow). Second, a real attribute of type Name with the name "name" needs to be defined at Customer which allows for assigning a name to a customer on M0 (cf. Figure 4). In conventional modeling, two attributes would be declared. The first attribute is defined within Name and will later contain the string "Otto". The second attribute will be defined within Customer and is of type Name, i.e., it will contain on M0 a reference to an instance of Name. However, this procedure is error prone since Name is assigned twice to Customer; first as a value within the collection attrs (from M2) and second as an attribute within the type facet of the clabject Customer. Also, an attribute for storing the value of an Attribute-instance is to be defined for each attribute. Materialization provides a solution to the first problem and Deep Instantiation to the second which are both shortly explained in the following paragraphs.

Materialization ultimately means that a new attribute is virtually created by assigning a value to an attribute. Therefore, the attrs Attribute on M2 contains a MMaterializationExtend instance which specifies that a new attribute should be created for each value put into the collection attrs. As name for the new attribute, the name of the Attribute-instance should be used. Besides that, MMaterializationExtend allows for specifying additional constraints like the visibility and multiplicity of the created element. Thus, after Name is put into the collection attrs at M1 a virtual attribute called "name" is created within the concept Customer.

Deep Instantiation in contrast allows for delaying the provision of values for attributes over levels; even though there exists an instanceOf relationship between two elements, attribute values might be assigned after two or even three instantiations. Therefore, an attribute that should later contain the real value at M0 can be defined at M2 with a deep instantiation counter value of 2. Each time, an instance is created, this counter (represented by an instance of the LMM type MDeepInstantiationExtend) is decremented. A value has to be assigned if this counter reaches 0 which happens on M0.

Therefore, the advanced modeling paradigms ease the creation of models by avoiding redundancies and/or complex type hierarchies. Further, the ability to change all levels of a meta hierarchy allows for easy and quick adaptation of existing models.

Creating such models was easy with OMME; also changing the content of any level within a hierarchy is possible and errors raised by such changes can be easily resolved. The only downside we experienced so far was that models on low levels look rather complex if one is not used to terms and notions of the LMM. Thus, we plan to implement the Dynamic API and make our text editor use that API. A step into this direction is the creation of wrappers for each level that allow for an easier programmatic access. So far creating an attribute and assigning it to a concept in code means to create an instance of an appropriate MAttribute, assign values for its attributes and add it to the collection of attributes of MConcept. Consequently, querying an attribute value, first the assignment that corresponds to the attribute has to be found. In order to ease this step, we defined a query language

which allows for quickly retrieving such information out of LMM models. This language is inspired by mSQL [27] but does not take over the complicated notion of mSQL and therefore is more "SQL like". However, this query language is not yet ported to "in memory models" such that a model first needs to be synchronized to a database. But on the bright side, all the issues we encountered so far are related to technical problems only which are mainly caused by the frameworks we use for implementation.

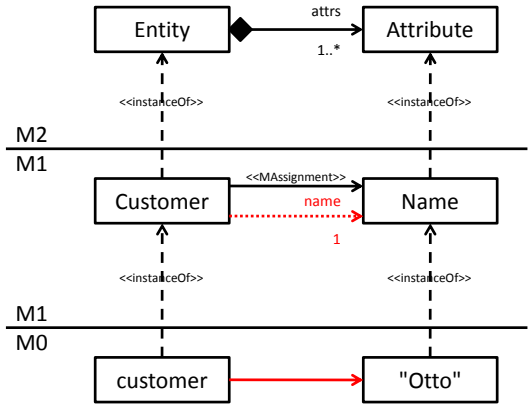

Figure 4. Outline of the meta hierarchy for defining ER models and instances of ER models

5. CONCLUSION

We presented a basic meta model within this publication that can serve as a basis for storing complete meta model hierarchies. This linguistic meta model covers aspects which are needed for implementing advanced modeling concepts such as (Extended) Powertypes, Deep Instantiation, Clabjects and Materialization. Along with the LMM we also introduced our prototypical implementation of the LMM that allows for creating models with an arbitrary hierarchy.

Beneficial for users is the ability to apply advanced modeling paradigms whose advantage are unification in treating model elements (with clabjects, no distinction into type and instance is necessary), reduction of complexity by increasing the expressiveness of the modeling language, avoidance of redundancies and the complex type hierarchies since e.g., powertypes can be at least partially resembled by a deep and complex inheritance hierarchy. Furthermore, because of the provision of one model for storing arbitrary hierarchies and the ability of tools to interpret these models accordingly, a set of "standard" tools is provided by OMME that can be used to create, modify and query models, editors and tools. The feasibility of providing such tools is given by our prototypical implementation.

6. REFERENCES

1. Object Management Group: UML 2.2 Super- & Infrastructure. http://www.omg.org/spec/UML/2.2/ (2010-08-10)
2. Fuentes-Fernández, L., Vallecillo-Moreno, A.: An Introduction to UML Profiles. UPGRADE - The European Journal for the Informatics Professional 5 (2004) 6-13
3. Odell, J.: Advanced Object-Oriented Analysis and Design Using UML. Cambridge University Press, New York, NY, USA (1998)
4. Jablonski, S., Volz, B., Dornstauder, S.: On the Implementation of Tools for Domain Specific Process Modelling. ENASE 2009, Milan, Italy (2009)
5. Atkinson, C., Kühne, T.: Meta-level Independent Modelling. Int'l Workshop Model Engineering 2000, Cannes, France
6. Dahchour, M., Pirotte, A., Zimányi, E.: Materialization and its Metaclass Implementation. IEEE Transctions on Knowledge and Data Engineering 14 (2002) 1078-1094
7. Atkinson, C., Kühne, T.: The Essence of Multilevel Metamodeling. 4th Int'l Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, Vol. 2185. Springer, Toronto, Canada (2001) 19-33
8. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling - A Foundation For Language Driven Development. CETEVA (2008)
9. Behrens, H., Clay, M., Efftinge, S., Eysholdt, M., Friese, P., Köhnlein, J., Wannheden, K., Zarnekow, S.: Xtext User Guide 1.0 (2010), Eclipse Foundation
10. Microsoft: Domain-Specific Language Tools. http://msdn.microsoft.com/en-us/library/bb126235.aspx (2010-08-10)
11. Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M., Eherer, S.: ConceptBase — A deductive object base for meta data management. Journal of Intelligent Information Systems 4 (1995) 167-192
12. de Lara, J., Vangheluwe, H.: Using ATOM3 as a Meta-Case Tool. ICEIS 2002, Ciudad Real, Spain
13. Davis, J.: GME: the generic modeling environment. 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. ACM, Anaheim, CA, USA (2003)
14. Object Management Group: MOF 2.0 Specification. http://www.omg.org/spec/MOF/2.0/ (2010-08-10)
15. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework. Addison-Wesley Longman, Amsterdam (2008)
16. Eclipse Foundation: Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/ (2010-08-10)
17. Atkinson, C., Kühne, T.: Concepts for Comparing Modeling Tool Architectures. Springer, (2005) 398-413
18. Gitzel, R., Hildenbrand, T.: A Taxonomy of Metamodel Hierarchies. Working paper, Chair in Information Systems III, University of Mannheim, Germany, http://bibserv7.bib.uni-mannheim.de/madoc/volltexte/2005/993/ (2010-08-10)
19. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java (TM) Language Specification. Addison-Wesley Professional (2005)
20. ECMA: ECMA-334: C# Language Specification. Geneva, Switzerland (2006)
21. Atkinson, C.: Meta-Modeling for Distributed Object Environments. EDOC '97, Gold Coast, Australia
22. Eclipse Foundation: Graphical Modeling Framework (GMF). http://www.eclipse.org/modeling/gmf/ (2010-08-10)
23. Eclipse Foundation: Eclipse Graphical Editing Framework (GEF). http://www.eclipse.org/gef/ (2010-08-10)
24. Jablonski, S., Bussler, C.: Workflow Management: Modeling Concepts, Architecture and Implementation. International Thomson Computer Press (1996)
25. Henderson-Sellers, B., Gonzalez-Perez, C.: Connecting Powertypes and Stereotypes. Journal of Object Technology (JOT) Vol. 4 (2005) 83-96
26. Henderson-Sellers, B., Gonzalez-Perez, C.: A Powertype-based Metamodelling Framework. SoSyM Vol.5 (2006) 72-90
27. Petrov, I.: Meta-data, Meta-Modelling and Query Processing in Meta-data Repository Systems. Shaker (2006)