KM3: A DSL for metamodel specification

Conference Paper *in* Lecture Notes in Computer Science · May 2006

DOI: 10.1007/11768869_14 · Source: DBLP

2 authors:

Frédéric Jouault
ESEO Group
169 PUBLICATIONS 7,074 CITATIONS
SEE PROFILE

Jean Bézivin
University of Nantes
181 PUBLICATIONS 7,744 CITATIONS
SEE PROFILE

KM3: a DSL for Metamodel Specification

Frédéric Jouault and Jean Bézivin

ATLAS team, INRIA and LINA
(frederic.jouault,jean.bezivin)@univ-nantes.fr

Abstract. We consider in this paper that a DSL (Domain Specific Language) may be defined by a set of models. A typical DSL is the ATLAS Transformation Language (ATL). An ATL program transforms a source model (conforming to a source metamodel) into a target model (conforming to a target metamodel). Being itself a model, the transformation program conforms to the ATL metamodel. The notion of metamodel is thus used to define the source DSL, the target DSL and the transformation DSL itself. As a consequence we can see that agility to define metamodels and precision of these definitions is of paramount importance in any model engineering activity. In order to fullfill the goals of agility and precision in the definition of our metamodels, we have been using a notation called KM3 (Kernel MetaMetaModel). KM3 may itself be considered as a DSL for describing metamodels. This paper presents the rationale for using KM3, some examples of its use and a precise definition of the language.

1 Introduction

Model engineering is strongly related to language engineering. Considering the important number of problem domains, there is a need for an equally important number of specialized languages. We have been using a language named KM3 (Kernel MetaMetaModel) to help defining these special purpose languages. This paper presents the rationale, semantics and other particularities of this language.

KM3 has its roots in the complex and evolving relations between modeling and visual languages. UML is a general purpose visual modeling language, but not every modeling language is a general purpose visual language. The OMG has proposed a language called MOF 2.0 [1] for the definition of its various metamodels (SPEM, UML, CWM, etc.). The problem was that there was no practical support environment for this language. As a replacement, the solution found was to use UML CASE tools for this purpose. The price to pay for this was an alignment of MOF with a subset of UML (mainly class diagrams). Since this time, the alignment has been more or less maintained through the various versions of UML and MOF. In other words, UML may be considered by certain as a multi-purpose language allowing defining software object-oriented terminal models and allowing also defining MOF metamodels. But this is not without drawbacks. When we need to build a metamodel (e.g. as source or target of a transformation), we have first to start building a UML class diagram, with certain properties.

The result is serialized in a first XMI file corresponding to the terminal model. It is then transformed into another XMI file corresponding to the metamodel. This conversion from a UML model to a MOF metamodel is called a *promotion* and implemented by some widely available tools like UML2MOF available in the NetBeans MDR [2] suite or also by an ATL [3] model transformation program.

We have experimented for some time with this approach. When the number of involved metamodels is limited (i.e. when one mainly deals with OMG fixed and stable metamodels), there are no major problems. But when we need multiple and evolving metamodels, we have found this approach to be very cumbersome. The only alternative has been to define KM3, a specialized textual language for specifying metamodels, including MOF metamodels. After experimenting with this language for two years, we are completely convinced of the practicality of the approach. Public libraries of more than one hundred metamodels expressed in KM3 are now available [4]. ATL, a QVT-like [5] model transformation language, uses KM3 natively to facilitate the handling of metamodels. Many other projects are also based on this format.

What remained to do is to establish a precise semantics for KM3. This is one of the objectives of the present work. Of course we have also to understand clearly the purpose and rationale of metamodel writing languages. In order to do so, we first need to define precisely what a metamodel exactly is. The definitions provided in this paper apply to the OMG MDA framework, but they are more general and may also correspond to several other technical spaces as defined in [6].

This paper is organized as follows. Section 2 provides the basic definitions related to models and DSLs. Section 3 provides an overview of KM3 including some current applications. Section 4 comes back on a more formal conceptual definition of KM3. A related work description is provided in section 5 before the conclusion.

2 Definitions

We consider models as the unifying concept in IT engineering. Models come in various flavors. A UML model, a Java program, an XML or RDF document, a database relational table, an entity-association schema are all examples of models. We call all of these λ-models where λ identifies a technical space [6] associated with a given precise metametamodel. A simple representation of terminal model, metamodel and metametamodel is given in Figure 1.

We may consider two main definitions of a model corresponding to its internal *organization* and its potential *utilization*. We choose to focus here on the *organization* of models. The study of model *utilization* and of its relations with model *organization* is out of the scope of this work. Then we give a definition of DSL and analyze the relations between DSLs and models.

2.1 Model Organization Definition

From an organization point of view, we propose the following definitions:

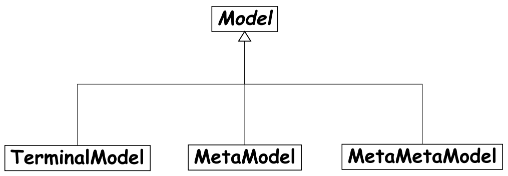


Fig. 1. General organization of a metamodeling stack

Definition 1. *A* directed multigraph $G = (N_G, E_G, \Gamma_G)$ *consists of a finite set of nodes* N_G*, a finite set of edges* E_G*, and a function* $\Gamma_G : E_G \rightarrow N_G \times N_G$ *mapping edges to their source and target nodes.*

Definition 2. *A* model $M = (G, \omega, \mu)$ *is a triple where:*

– $G = (N_G, E_G, \Gamma_G)$ *is a directed multigraph,*
– ω *is itself a model (called the reference model of M) associated to a graph* $G_\omega = (N_\omega, E_\omega, \Gamma_\omega)$*,*
– $\mu : N_G \cup E_G \rightarrow N_\omega$ *is a function associating elements (nodes and edges) of* G *to nodes of* G_ω*.*

Remarks. The relation between a model and its reference model is called conformance and is noted *conformsTo* or abbreviated in *c2* throughout this paper. Elements of ω are called metaelements. μ is neither injective (several model elements may be associated to the same metaelement) nor surjective (not all metaelements need to be associated to a model element).

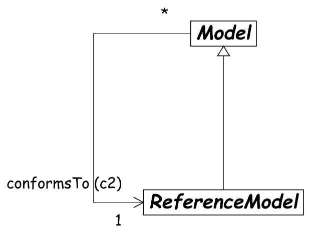


Fig. 2. Definition of *model* and *reference model*

Figure 2 illustrates definition 2. The definition of model given above allows for an indefinite number of upper modeling layers. For practical purpose, we need to stop at some level. We observe that only three levels are used in several technical spaces:

– In *XML*: documents, schemas and the schemas of XML Schema for XML,
– In *EBNF*: programs, grammars and the grammar of EBNF.

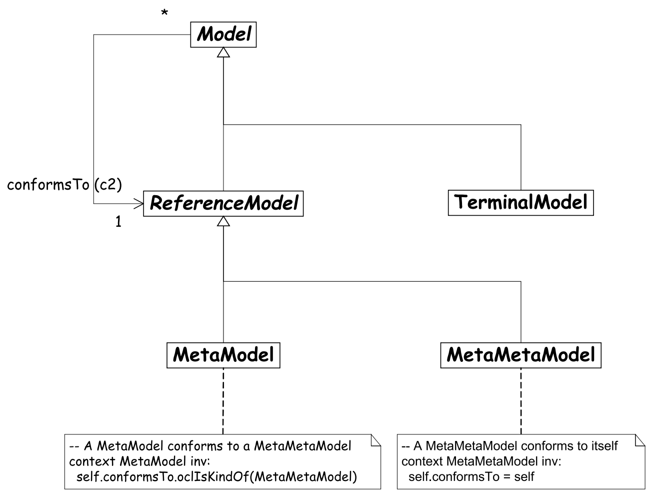
We call these levels: M1, M2 and M3. M1 consists of all models that are not metamodels. M2 consists of all metamodels that are not the metametamodel. M3 consists of a unique metametamodel for each given technical space. We may now proceed to giving additional definitions.

Definition 3. *A* metametamodel *is a model that is its own reference model (i.e. it conforms to itself).*

Definition 4. *A* metamodel *is a model such that its reference model is a metametamodel.*

Definition 5. *A* terminal model *is a model such that its reference model is a metamodel.*

Figure 3 shows how to adapt the definition of model to this three-level modeling stack. The structure for models defined in this section is compatible with the OMG view as illustrated in the MDA guide [7].



```
-- A MetaModel conforms to a MetaMetaModel
context MetaModel inv:
  self.conformsTo.oclIsKindOf(MetaMetaModel)
```

```
-- A MetaMetaModel conforms to itself
context MetaMetaModel inv:
  self.conformsTo = self
```

Fig. 3. Metamodeling stack representation with model definition

2.2 Domain Specific Language

Language engineering is at the hearth of computer science. There are a variety of categories of languages. We discuss here only a small facet of language engineering. A distinction is often made between programming languages and modeling languages. Typical examples are PL/1 and UML. The distinction between these categories has mainly to do with canonical executability and is currently much evolving. Another distinction is between General Purpose Languages (GPLs) and Domain Specific Languages (DSLs). PL/1 and UML are two examples of GPLs. R [8], SQL [9] or Excel are examples of DSLs. Java and C# are examples of general purpose programming languages.

We also understand that the distinction between GPLs and DSLs is orthogonal to many other language classifications. For example there are indifferently visual or textual GPLs or DSLs. Similarly DSLs and GPLs may fall under various categories of being object-oriented, event-oriented, rule-oriented, function-oriented, etc. There are examples of imperative and declarative GPLs and DSLs as well.

A DSL is a language designed to be useful for a limited set of tasks, in contrast to general-purpose languages that are supposed to be useful for much more generic tasks, crossing multiple application domains. A typical example of DSL is GraphViz [10], a language used to define directed graphs, which creates a visual representation of that graph as a result. Some GPLs have started as DSLs and have sometimes evolved towards genericity to become GPLs. The reverse process has not been observed in the history of programming languages.

Like many other languages, DSLs have many common properties [11]:

- They have usually a concrete syntax
- They may also have an abstract syntax
- They have a semantics, implicitly or explicitly defined

Of course there are several ways to define these syntax and semantics. The most known are grammar-based systems.

2.3 DSLs and Models

There are strong relations between DSLs and models. We discuss here the possibility of using model-based solutions for defining the syntax and semantics of DSLs.

Definition 6. *A DSL is a set of coordinated models.*

Each model in this set contributes to a part of its definition. A given model may, for instance, specify one of the following aspects:

- **Domain definition metamodel.** One of the defining entities of a DSL is a Domain Definition MetaModel (DDMM). It introduces the basic entities of the domain and their mutual relations. This *base ontology* plays a central role in the definition of the DSL. For example, a DSL for directed graph manipulation will contain the concepts of nodes and edges, and will state that an edge may connect a source node to a target node. Such a DDMM plays the role of the abstract syntax for a DSL.
- **Concrete syntaxes.** A DSL may have different concrete syntaxes. Each one is defined by a transformation model mapping the DDMM onto a *display surface* metamodel. Examples of display surface metamodels may be SVG or DOT [10], but also XML. An example of such a transformation for a Petri

net DSL is the mapping from places to circles, from transitions to rectangles and from arcs to arrows. The display surface metamodel will then have the concepts of Circle, Rectangle and Arrow.
- **Execution semantics.** A DSL may have an execution semantics definition. This semantics definition is also defined by a transformation model mapping the DDMM onto another DSL having itself an execution semantics or even to a GPL. The firing rules of a Petri net may for example be mapped into a Java code model.
- **Other operations on DSLs.** In addition to canonical execution, there are plenty of other possible operations on programs based on a given DSL. Each may be defined by a similar mapping represented by a transformation model. For example if one wishes to query DSL programs, a standard mapping of the DDMM onto Prolog may be useful. The study of these other operations on DSLs is an open research subject.

3 KM3 Overview

3.1 Description

The purpose of KM3 is to give a relatively simple solution to define the Domain Definition MetaModel of a DSL. KM3 is therefore a Domain Specific Language to define metamodels:

- **Domain definition metamodel.** The DDMM of KM3 is a metametamodel, to which other DDMMs conform. This DDMM may be defined in KM3 (see appendix A.1), just like EBNF (a notation to define grammars) may be described in EBNF using only a few lines. It uses concepts like *Class*, *Attribute*, and *Reference*. It is structurally close to eMOF 2.0 [1] and Ecore [12].
- **Concrete syntax.** A default textual concrete syntax has been defined for KM3 (see appendix A.2)). This allows straightforward definitions of metamodels with any text editor.
- **Semantics.** The semantics of KM3 enables the specification of metamodels and models according to the definitions given in section 2. A precise conceptual definition of KM3 is presented in section 4. Mappings to and from MOF 1.4 [13] and Ecore have notably been defined in ATL, making KM3 usable with tools like Eclipse EMF [12] and Netbeans MDR.

As a metametamodel, KM3 is simpler than MOF 1.4, MOF 2.0 [1] and Ecore. It contains only 14 classes whereas, for instance, Ecore has 18 classes and MOF 1.4 has 28 classes. Only the core concepts of these other metametamodels are available in KM3.

Figure 4 describes an XML metamodel in the standard visual notation of class diagrams. This XML metamodel corresponds to the following KM3 description:

```
package XML {
  abstract class Node {
    attribute name : String;
```

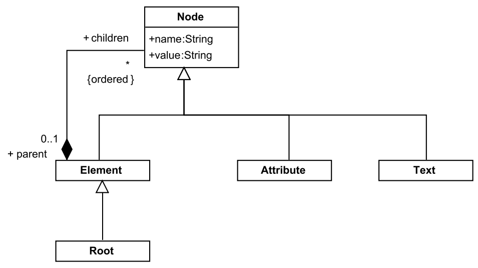


Fig. 4. Visual presentation of an XML metamodel

```
    attribute value : String;
    reference parent[0-1] : Element oppositeOf children;
  }

  class Attribute extends Node {}

  class Text extends Node {}

  class Element extends Node {
    reference children[*] ordered container : Node oppositeOf parent;
  }

  class Root extends Element {}
}

package PrimitiveTypes {
  datatype Boolean;
  datatype Integer;
  datatype String;
}
```

3.2 Applications

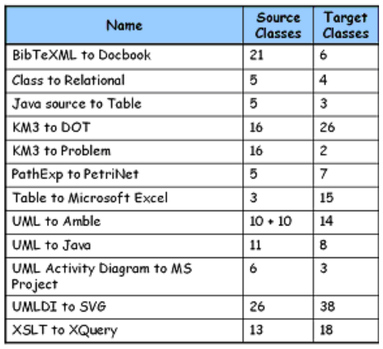
KM3 has been defined as an answer to frequent requests of users that were defining model transformations in the ATL language. In principle source and target metamodels for QVT-like transformations should be written in XMI. When the transformation is based on standard metamodels like UML metamodels, the XMI serialization of these metamodels may be found on the OMG site and there is no need for any additional formalism.

The practice of model transformation, with a growing community of ATL users, has however obliged to amend this opinion. During the development of these transformations, it became clear that very often the standard metamodels were not sufficient and that many of the transformations needed specific metamodels. Furthermore, the definition of these metamodels is often an iterative process involving a progressive elaboration.

In order to illustrate this, we provide below some examples of transformations written in ATL. The complete code and documentation of these transformations may be found in the open source library of transformation available on [14] and [15].

- *Ant2Maven* and *Make2Ant* are partial transformations between well known software engineering build tools (Make, Ant and Maven).
- *BibTeX2DocBook* is a transformation of a BibTeXML model to a DocBook composed document.
- The *JavaSource2Table* example computes a static call graph of a Java program and presents it in a tabular style. From there, one may use the XHTML or the Excel metamodels to project to other display surfaces, by transformation chaining.
- The *KM32DOT* allows drawing graphical presentations of metamodels. DOT is an automatic graph layout program from GraphViz [10]. The aim of this transformation is to generate a visualization, in the form of a class diagram, of any KM3 metamodel by automatic layout
- The *UMLActivityDiagram2MSProject* example describes a transformation from a loop free UML activity diagram (describing some tasks series) to MS Project. The transformation is based on a simplified subset of the UML State Machine metamodel. This transformation produces a project defined in conformance to a limited subset of the MSProject metamodel.

The following table (Figure 5) gives another sample from the same model transformation library, where the numbers of classes in the source and target metamodels are provided. Without describing in detail all these transformations, it becomes clear that most source and target metamodels have to be defined and even in the case they are standard (like the UML activity diagram), they often correspond to a small subset of the standard metamodel.



Name	Source Classes	Target Classes
BibTeXML to Docbook	21	6
Class to Relational	5	4
Java source to Table	5	3
KM3 to DOT	16	26
KM3 to Problem	16	2
PathExp to PetriNet	5	7
Table to Microsoft Excel	3	15
UML to Amble	10 + 10	14
UML to Java	11	8
UML Activity Diagram to MS Project	6	3
UMLDI to SVG	26	38
XSLT to XQuery	13	18

Fig. 5. A sample of transformations from the ATL library

As a consequence, the definition of source and target metamodels in a transformation is an important part of the design of this transformation. We need a notation that will allow easy and precise definition and modification of these metamodels. Even if this seems counter intuitive, users have been asking for textual languages instead of visual languages for performing this task.

The KM3 language has been very useful in supporting rapid and precise definition of metamodels for various situations. When studying the interoperability between several tools (like Bugzilla, Make, MS Project, or Mantis), the data models of these tools are usually captured in a metamodel, and the bridges may be designed as transformations, directly using these metamodels.

We have previously mentioned the initial library of ATL transformations. What is also interesting is that a significant library of the corresponding metamodels has also grown in the same time and may be found at [16]. There are many issues that can be studied on the basis of this initial library. The first one is related to reusability of these metamodels. More important questions may be raised on the various relations that may hold between these metamodels and also to the metadata about them.

4 Conceptual Definition of KM3

Definition 7. *A* KM3-model *is a model defined using KM3 as a metametamodel.*

This section only deals with KM3-models. Therefore, we use model to mean KM3-model. We present here a formal specification of KM3 based on first order logic. Only metamodels, not terminal models, may conform to KM3. However, KM3 semantics also impacts terminal models by constraining them according to their reference models. Two main predicates are used to define KM3-models, including the KM3 metametamodel itself. For a model M (see definition 2), we define:

– $Node(x, y)$**.** This predicate states that a node $x \in N_G$ is associated to a node $y \in N_\omega$ by the function μ.
– $Edge(x, y, z)$**.** This predicate states that an edge between node $x \in N_G$ and node $y \in N_G$ is associated to a node $z \in N_\omega$ by the function μ. In KM3, multiple edges between two given nodes may only exist if their associated metaelements are distinct. Therefore, the triple (x, y, z) uniquely identifies an edge.

Formulas are used to express constraints on KM3-models. We start by defining a simplified version of KM3 called *SimpleKM3* with only classes and references. Then we introduce additional concepts: opposite references and inheritance.

4.1 Definition of SimpleKM3

SimpleKM3 is a simplified version of KM3 using only classes and references. A visual representation of *SimpleKM3* is given in Figure 6. Figure 7 gives the formal definition of *SimpleKM3*. There are only two classes: *class* (line 1) and *reference* (line 2). There are two references: *features* (line 3) and *type* (line 4). The *features* reference connects a class to its references (lines 5 and 6). The *type* reference connects a reference to its type (lines 7 and 8).

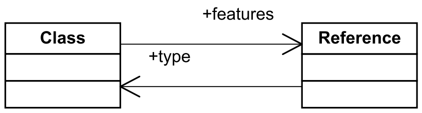


Fig. 6. Class diagram representation of *SimpleKM3*

1. $Node(\text{class}, \text{class})$	5. $Edge(\text{class}, \text{features}, \text{features})$
2. $Node(\text{reference}, \text{class})$	6. $Edge(\text{features}, \text{reference}, \text{type})$
3. $Node(\text{features}, \text{reference})$	7. $Edge(\text{reference}, \text{type}, \text{features})$
4. $Node(\text{type}, \text{reference})$	8. $Edge(\text{type}, \text{class}, \text{type})$

Fig. 7. Formal definition of *SimpleKM3*

We define a new predicate $IsKindOf(x, y)$, which is for now equivalent to predicate $Node(x, y)$:

$$\forall xy IsKindOf(x, y) \leftrightarrow Node(x, y) \tag{1}$$

It will be redefined in section 4.3 when we introduce class inheritance in *SimpleKM3*. We still use the $Node(x, y)$ predicate to define nodes but use this new predicate in formulas that are also valid for subclasses. This is the case for formulas (5) and (6).

A *SimpleKM3*-model (i.e. model, metamodel or metametamodel) is valid if the following formulas are verified:

- **Metaelement uniqueness.** μ, as a function, can only associate a single metaelement to a given model node.

$$\forall xyz Node(x, y) \wedge Node(x, z) \rightarrow y = z \tag{2}$$

  There is no similar formula for edges because there may be several edges of different types between two given nodes.
- **Node metaelelements are classes.** Any node that is used as a metaelement of another node must have node *class* as its metaelement.

$$\forall xy Node(x, y) \rightarrow Node(y, \text{class}) \tag{3}$$

- **Edge metaelements are references.** An edge can only exists between nodes and must have node $reference$ as its type.

$$\forall xyz Edge(x, y, z) \rightarrow (\exists x_t Node(x, x_t)) \wedge (\exists y_t Node(y, y_t)) \wedge Node(z, \text{reference}) \tag{4}$$

- **Edge target.** An edge typed by reference z can only target a node typed y_t if the type of z is y_t.

$$\forall xyz Edge(x, y, z) \rightarrow (\exists y_t IsKindOf(y, y_t) \wedge Edge(z, y_t, \text{type})) \tag{5}$$

– **Edge source.** An edge typed by reference z can only have a node typed x_t as source if z is a feature of x_t.

$$\forall xyz Edge(x, y, z) \rightarrow (\exists x_t IsKindOf(x, x_t) \wedge Edge(x_t, z, \text{features})) \qquad (6)$$

– **Reference type uniqueness.** A reference has a unique type.

$$\forall xyz Edge(x, y, \text{type}) \wedge Edge(x, z, \text{type}) \rightarrow y = z \qquad (7)$$

We must specify this constraint in *SimpleKM3* because it does not have the concept of multiplicity.

4.2 Adding Opposite References

Opposite references work in pairs. They are especially convenient to enable bidirectional navigation. For instance, in our first version of *SimpleKM3*, although we can get the features of a class, we cannot get the class owning a given reference. Figure 8 defines the *opposite* reference belonging to and targeting the *reference* class.

9. $Node(\text{opposite}, \text{reference})$
10. $Edge(\text{reference}, \text{opposite}, \text{features})$
11. $Edge(\text{opposite}, \text{reference}, \text{type})$

Fig. 8. Addition of opposite reference to *SimpleKM3*

A *SimpleKM3*-model (i.e. model, metamodel or metametamodel) with *opposite* is valid if the following formulas are verified:

– **Opposite uniqueness.** A reference has at most one opposite.

$$\forall xyz Edge(x, y, \text{opposite}) \wedge Edge(x, z, \text{opposite}) \rightarrow y = z \qquad (8)$$

– **References work in pairs.**

$$\forall xy Edge(x, y, \text{opposite}) \rightarrow Edge(y, x, \text{opposite}) \qquad (9)$$

– **Opposite references have opposite extremities.**

$$\forall xyz Edge(x, y, \text{opposite}) \wedge Edge(z, x, \text{features}) \rightarrow Edge(y, z, \text{type}) \qquad (10)$$

We can now extend *SimpleKM3* with an *owner* reference opposite to the *features* reference as shown on Figure 9. The resulting definition of *SimpleKM3* corresponds to the class diagram given in Figure 10. It is now possible to navigate from *reference* to *class*.

12. $Node(\text{owner}, \text{reference})$
13. $Edge(\text{reference}, \text{owner}, \text{features})$
14. $Edge(\text{owner}, \text{class}, \text{type})$
15. $Edge(\text{owner}, \text{features}, \text{opposite})$
16. $Edge(\text{features}, \text{owner}, \text{opposite})$

Fig. 9. Addition of some opposite references to *SimpleKM3*

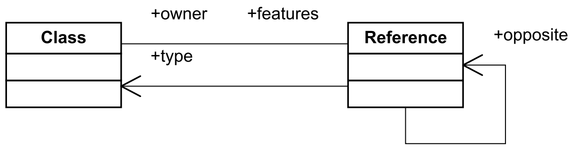


Fig. 10. Class diagram representation of *SimpleKM3* with opposites

4.3 Adding Inheritance

In KM3, inheritance allows reuse of references defined in supertypes. Overriding of inherited features is not allowed. Figure 11 introduces the *supertypes* reference from *class* to *class*. Figure 12 gives the class diagram of *SimpleKM3* with inheritance. In order to be able to use inherited references or to define edges targeting subclasses of a reference type, we redefine $IsKindOf(x)$ (see formula 1) accordingly:

$$\forall xy IsKindOf(x,y) \leftrightarrow Node(x,y) \vee (\exists z Node(x,z) \wedge ConformsTo(z,y)) \quad (11)$$

This new definition makes use of the $ConformsTo(x,y)$ predicate, recursively defined as follows:

$$\begin{aligned} \forall xy ConformsTo(x,y) \leftrightarrow (x=y)\vee \\ (\exists z Edge(x,z,\text{supertypes}) \wedge ConformsTo(z,y)) \end{aligned} \quad (12)$$

Circular inheritance is forbiden. The $ConformsTo(x,y)$ predicate could not be defined otherwise. With this new definitions, formulas (6) and (5) remain valid.

17. $Node(\text{supertypes}, \text{reference})$
18. $Edge(\text{class}, \text{supertypes}, \text{features})$
19. $Edge(\text{supertypes}, \text{class}, \text{type})$

Fig. 11. Addition of inheritance to *SimpleKM3*

4.4 Other KM3 Concepts

We defined the formal semantics of the remaining KM3 concepts as well: packages, class abstractness, data types, attributes, enumerations, reference containment, multiplicity, etc. However, they do not fit in this paper because of space

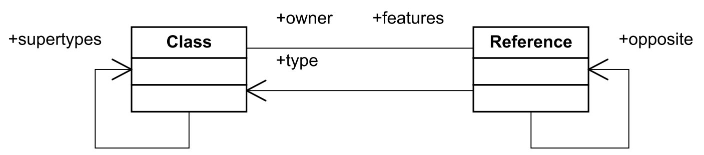


Fig. 12. Class diagram representation of *SimpleKM3* with opposites and inheritance

limitation. A complete specification of KM3 in Prolog is available in appendix A.3. This program uses the same predicates we defined in this section plus the $Prop(x, y, z)$ predicate where $x \in N_G$, $y \in N_\omega$ is an *attribute*, and z is a value. We do not further detail this predicate, which is used as a shortcut to avoid representing primitive values as nodes explicitly. The set of constraints implemented in the program is illustrative of the characterization of KM3. We do not claim completness here.

5 Related Work

Other modeling frameworks offer capabilities similar to those of KM3:

- **OMG MOF.** MOF is a standard metametamodel from OMG, of which there exist several versions (e.g. MOF 1.4 [13] and MOF 2.0 [1]). All of them are more complex than KM3 (i.e. they contain more classes, see section 3.1). None has a formal semantics. Their standard concrete syntax is XMI, which is based on XML and is, as such, more verbose than KM3. As noted in section 3.1, we have defined ATL transformations from MOF 1.4 to KM3 and from KM3 to MOF 1.4.
- **HUTN.** Human Usable Textual Notation [17] (HUTN) is a standard by OMG to give a default textual notation to each metamodel. Because it is an automatic mapping from MOF to EBNF, it is more verbose than KM3.
- **Eclipse EMF Ecore.** Ecore [12] is a metametamodel close to MOF 2.0 but with a standard textual notation: emfatic. One difference with KM3 is that emfatic provides EMF-specific constructs (e.g. to customize Java code generation). One of our experiments has shown that such additional information may be embedded into KM3 comments. Another difference is that Ecore has no formal semantics. As noted in section 3.1, we have defined ATL transformations from Ecore to KM3 and from KM3 to Ecore.
- **Typed graphs.** Typed Attributed Graphs [18] are the conceptual framework on which graph transformation is based. They have a precise formal semantics. In opposition to KM3 and the definitions given in section 2, there is no explicit metametamodel: type graphs are not themselves typed.
- **sNets.** sNets [19] are one of our past experiments. We have learnt much from them and KM3 is based on this knowledge. One difference with KM3 is that there is an explicit representation of μ in the sNet metametamodel.

However this may lead to using hypergraphs to provide a complete general solution, with possible strong constraints on implementation overhead.

6 Conclusions

In this paper we have proposed a metamodel definition language. We have seen other possibilities of DSLs for performing such tasks like XMI or Emfatic. Each DSL has some specificities, some advantages and drawbacks. For Emfatic for example, the projection to Java is an important feature; for XMI, the possibility to take into account terminal models as well as metamodels is an essential property.

The KM3 language is intended to be a lightweight textual metamodel definition language allowing easy creation and modification of metamodels. The metamodels expressed in KM3 have good readibility properties. The formalism is sufficiently rich to support essential information. Additional information can be expressed as metadata pragmas not described here. Metamodels expressed in KM3 may be easily converted to/from other notations like Emfatic or XMI.

Among the properties of KM3 is the possibility to use it for the definition of non-MOF based models. KM3 has also been designed to cross technical spaces.

The contribution of this paper is a clean semantics for a metamodel definition language. To the best of our knowledge, such a definition has not been proposed for such a language. As a side effect of this work, we have been able to give a precise and original definition of a model, in the context of multiple technical spaces. All the tools currently available in the ATLAS Model Management Platform [15] are completely based on this operational definition.

7 Acknowledgements

This work has been partially supported by ModelWare, IST European project 511731. We thank Ivan Kurtev and all the members of the ATLAS team for their support to this work.

References

1. OMG: Meta Object Facility (MOF) 2.0 Core Specification, OMG Document ptc/03-10-04, `http://www.omg.org/docs/ptc/03-10-04.pdf`. (2003)
2. netBeans.org: Netbeans Meta Data Repository (MDR), `http://mdr.netbeans.org/`. (2006)
3. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Satellite Events at the MoDELS 2005 Conference. Volume 3844 of Lecture Notes in Computer Science., Springer-Verlag (2006) 128–138
4. ATLAS team: ATLAS MegaModel Management (AM3) Home page, `http://www.eclipse.org/gmt/am3/`. (2006)
5. OMG: MOF QVT Final Adopted Specification, OMG Document ptc/2005-11-01, `http://www.omg.org/docs/ptc/05-11-01.pdf`. (2005)

6. Bézivin, J., Kurtev, I.: Model-based technology integration with the technical space concept. In: Proceedings of the Metainformatics Symposium, Springer-Verlag (2005)
7. Object and Reference Model Subcommittee (ORMSC) of the OMG Architecture Board: A Proposal for an MDA Foundation Model, white paper OMG-ORMSC/05-08-01, `http://www.omg.org/cgi-bin/doc?ormsc/05-08-01`. (2005)
8. Bates, D., et al.: R Language Definition, `http://stat.ethz.ch/R-manual/R-patched/doc/manual/R-lang.html`. (2006)
9. McJones, P.R., ed.: The 1995 SQL Reunion: People, Project, and Politics, May 29, 1995. Volume SRC1997-018. (1997)
10. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Software — Practice and Experience **30**(11) (2000) 1203–1233
11. Harel, D., Rumpe, B.: Meaningful modeling: What's the semantics of "semantics"? Computer **37**(10) (2004) 64–72
12. Budinsky, F., Steinberg, D., Ellersick, R., Merks, E., Brodsky, S.A., Grose, T.J.: Eclipse Modeling Framework. Addison Wesley (2003)
13. OMG: Meta Object Facility (MOF) Specification, version 1.4, OMG Document formal/2002-04-03, `http://www.omg.org/technology/documents/formal/mof.htm`. (2002)
14. Eclipse Foundation: Generative Model Transformer (GMT) Home page, `http://www.eclipse.org/gmt/`. (2006)
15. ATLAS team: ATLAS Transformation Language (ATL) Home page, `http://www.eclipse.org/gmt/atl/`. (2006)
16. ATLAS team: Atlantic Metamodel Zoo, `http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/`. (2006)
17. OMG: Human-Usable Textual Notation, v1.0, OMG Document formal/04-08-01, `http://www.omg.org/technology/documents/formal/hutn.htm`. (2004)
18. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Graph Transformations: Second International Conference, ICGT 2004. Volume 3256 of Lecture Notes in Computer Science., Springer-Verlag (2004) 161–177
19. Bézivin, J.: sNets: A first generation model engineering platform. In: Satellite Events at the MoDELS 2005 Conference. Volume 3844 of Lecture Notes in Computer Science., Springer-Verlag (2006) 169–181

Appendix A

A.1 KM3 Definition of KM3

```
package KM3 {
  abstract class ModelElement {
    attribute name : String;
    reference "package" : Package oppositeOf contents;
  }

  class Classifier extends ModelElement {}

  class DataType extends Classifier {}

  class Enumeration extends Classifier {
    reference literals[*] ordered container : EnumLiteral oppositeOf
        ↪enum;
  }

  class EnumLiteral extends ModelElement {
    reference enum : Enumeration oppositeOf literals;
  }

  class Class extends Classifier {
    attribute isAbstract : Boolean;
    reference supertypes[*] : Class;
    reference structuralFeatures[*] ordered container :
        ↪StructuralFeature oppositeOf owner;
  }

  class TypedElement extends ModelElement {
    attribute lower : Integer;
    attribute upper : Integer;
    attribute isOrdered : Boolean;
    attribute isUnique : Boolean;
    reference type : Classifier;
  }

  class StructuralFeature extends TypedElement {
    reference owner : Class oppositeOf structuralFeatures;
    reference subsetOf[*] : StructuralFeature oppositeOf derivedFrom;
    reference derivedFrom[*] : StructuralFeature oppositeOf subsetOf;
  }

  class Attribute extends StructuralFeature {}

  class Reference extends StructuralFeature {
    attribute isContainer : Boolean;
    reference opposite[0-1] : Reference;
  }

  class Package extends ModelElement {
    reference contents[*] ordered container : ModelElement oppositeOf "
        ↪package";
    reference metamodel : Metamodel oppositeOf contents;
  }

  class Metamodel extends LocatedElement {
    reference contents[*] ordered container : Package oppositeOf
        ↪metamodel;
  }
}

package PrimitiveTypes {
  datatype Boolean;
  datatype Integer;
  datatype String;
}
```

A.2 Formal Syntax of KM3

package	→ `package` *name* { *classifiers* }
classifiers	→
classifiers	→ *classifier classifiers*
classifier	→ *class*
classifier	→ *datatype*
classifier	→ *enumeration*
class	→ *isabstract* `class` *name supertypes* { *features* }
isabstract	→
isabstract	→ `abstract`
supertypes	→
supertypes	→ `extends` *typelist*
typelist	→ *typeref*
typelist	→ *typeref*, *typelist*
features	→
features	→ *feature features*
feature	→ *attribute*
feature	→ *reference*
attribute	→ `attribute` *name multiplicity* `:` *typeref* `;`
reference	→ `reference` *name multiplicity iscontainer* `:` *typeref* `oppositeOf` *name* `;`
multiplicity	→ *bounds*
multiplicity	→ *bounds* `ordered`
bounds	→
bounds	→ `[` *integer* `-` *integer* `]`
bounds	→ `[` *integer* `- * ]`
bounds	→ `[ * ]`
iscontainer	→
iscontainer	→ `container`
datatype	→ `datatype` *name* `;`
enumeration	→ `enumeration` *name* `{` *literals* `}`
literals	→
literals	→ *literal literals*
literal	→ `literal` *name* `;`
typeref	→ *name*

A.3 Prolog Definition of KM3

```
/* KM3 metametamodel definition */

node(km3,package).
prop(km3,name,"KM3").
edge(km3,modelElement,contents).
edge(modelElement,km3,me_package).
edge(km3,package,contents).
edge(package,km3,me_package).
edge(km3,classifier,contents).
edge(classifier,km3,me_package).
edge(km3,dataType,contents).
edge(dataType,km3,me_package).
edge(km3,class,contents).
edge(class,km3,me_package).
edge(km3,structuralFeature,contents).
edge(structuralFeature,km3,me_package).
edge(km3,reference,contents).
edge(reference,km3,me_package).
edge(km3,attribute,contents).
edge(attribute,km3,me_package).
edge(km3,boolean,contents).
edge(boolean,km3,me_package).
edge(km3,integer,contents).
edge(integer,km3,me_package).
edge(km3,string,contents).
edge(string,km3,me_package).

/* class ModelElement */
node(modelElement,class).
```

```
prop(modelElement,name,"ModelElement").
prop(modelElement,isAbstract,true).

/* attribute ModelElement.name : String */
node(name,attribute).
prop(name,name,"name").
prop(name,lower,1).
prop(name,upper,1).
prop(name,isOrdered,false).
prop(name,isUnique,false).
edge(modelElement,name,features).
edge(name,modelElement,owner).
edge(name,string,type).

/* reference ModelElement.package : Package oppositeOf contents */
node(me_package,reference).
prop(me_package,name,"package").
prop(me_package,isContainer,false).
prop(me_package,lower,0).
prop(me_package,upper,1).
prop(me_package,isOrdered,false).
prop(me_package,isUnique,false).
edge(modelElement,me_package,features).
edge(me_package,modelElement,owner).
edge(me_package,package,type).
edge(me_package,contents,opposite).

/* class Package extends ModelElement */
node(package,class).
prop(package,name,"Package").
prop(package,isAbstract,false).
edge(package,modelElement,supertypes).

/* reference Package.contents[*] ordered container : ModelElement oppositeOf package */
node(contents,reference).
prop(contents,name,"contents").
prop(contents,isContainer,true).
prop(contents,lower,0).
prop(contents,upper,-1).
prop(contents,isOrdered,false).
prop(contents,isUnique,false).
edge(package,contents,features).
edge(contents,package,owner).
edge(contents,modelElement,type).
edge(contents,me_package,opposite).

/* class Classifier extends ModelElement */
node(classifier,class).
prop(classifier,name,"Classifier").
prop(classifier,isAbstract,true).
edge(classifier,modelElement,supertypes).

/* class DataType extends Classifier */
node(dataType,class).
prop(dataType,name,"DataType").
prop(dataType,isAbstract,false).
edge(dataType,classifier,supertypes).

/* class Class extends Classifier */
node(class,class).
prop(class,name,"Class").
prop(class,isAbstract,false).
edge(class,classifier,supertypes).

/* attribute Class.isAbstract : Boolean */
node(isAbstract,attribute).
prop(isAbstract,name,"isAbstract").
prop(isAbstract,lower,1).
prop(isAbstract,upper,1).
prop(isAbstract,isOrdered,false).
prop(isAbstract,isUnique,false).
edge(class,isAbstract,features).
edge(isAbstract,class,owner).
edge(isAbstract,boolean,type).

/* reference Class.features[*] ordered container : StructuralFeature oppositeOf owner */
node(features,reference).
prop(features,name,"structuralFeatures").
prop(features,isContainer,true).
prop(features,lower,0).
prop(features,upper,-1).
prop(features,isOrdered,true).
prop(features,isUnique,true).
edge(class,features,features).
edge(features,class,owner).
edge(features,structuralFeature,type).
edge(features,owner,opposite).

/* reference Class.supertypes[*] : Class */
node(supertypes,reference).
prop(supertypes,name,"supertypes").
prop(supertypes,isContainer,false).
prop(supertypes,lower,0).
prop(supertypes,upper,-1).
prop(supertypes,isOrdered,false).
prop(supertypes,isUnique,true).
edge(class,supertypes,features).
edge(supertypes,class,owner).
edge(supertypes,class,type).

/* abstract class StructuralFeature extends ModelElement */
node(structuralFeature,class).
prop(structuralFeature,name,"StructuralFeature").
prop(structuralFeature,isAbstract,true).
edge(structuralFeature,modelElement,supertypes).

/* attribute StructuralFeature.lower : Integer */
node(lower,attribute).
prop(lower,name,"lower").
prop(lower,lower,1).
prop(lower,upper,1).
prop(lower,isOrdered,false).
prop(lower,isUnique,false).
edge(structuralFeature,lower,features).
edge(lower,structuralFeature,owner).
edge(lower,integer,type).

/* attribute StructuralFeature.upper : Integer */
node(upper,attribute).
prop(upper,name,"upper").
prop(upper,lower,1).
prop(upper,upper,1).
prop(upper,isOrdered,false).
prop(upper,isUnique,false).
edge(structuralFeature,upper,features).
edge(upper,structuralFeature,owner).
edge(upper,integer,type).

/* attribute StructuralFeature.isOrdered : Boolean */
node(isOrdered,attribute).
prop(isOrdered,name,"isOrdered").
prop(isOrdered,lower,1).
prop(isOrdered,upper,1).
prop(isOrdered,isOrdered,false).
prop(isOrdered,isUnique,false).
edge(structuralFeature,isOrdered,features).
edge(isOrdered,structuralFeature,owner).
edge(isOrdered,boolean,type).

/* attribute StructuralFeature.isUnique : Boolean */
node(isUnique,attribute).
prop(isUnique,name,"isUnique").
prop(isUnique,lower,1).
prop(isUnique,upper,1).
prop(isUnique,isOrdered,false).
prop(isUnique,isUnique,false).
edge(structuralFeature,isUnique,features).
edge(isUnique,structuralFeature,owner).
edge(isUnique,boolean,type).

/* reference StructuralFeature.owner : Class oppositeOf features */
node(owner,reference).
prop(owner,name,"owner").
prop(owner,isContainer,false).
prop(owner,lower,1).
prop(owner,upper,1).
prop(owner,isOrdered,false).
prop(owner,isUnique,false).
edge(structuralFeature,owner,features).
edge(owner,structuralFeature,owner).
edge(owner,class,type).
edge(owner,features,opposite).

/* reference StructuralFeature.type : Class */
node(type,reference).
prop(type,name,"type").
prop(type,isContainer,false).
prop(type,lower,1).
prop(type,upper,1).
prop(type,isOrdered,false).
prop(type,isUnique,false).
edge(structuralFeature,type,features).
edge(type,structuralFeature,owner).
edge(type,classifier,type).

/* class Reference extends StructuralFeature */
node(reference,class).
prop(reference,name,"Reference").
prop(reference,isAbstract,false).
edge(reference,structuralFeature,supertypes).

/* attribute Reference.isContainer : Boolean */
node(isContainer,attribute).
prop(isContainer,name,"isContainer").
prop(isContainer,lower,1).
prop(isContainer,upper,1).
prop(isContainer,isOrdered,false).
prop(isContainer,isUnique,false).
edge(reference,isContainer,features).
edge(isContainer,reference,owner).
edge(isContainer,boolean,type).

/* reference Reference.opposite : Reference */
node(opposite,reference).
prop(opposite,name,"opposite").
prop(opposite,isContainer,false).
prop(opposite,lower,0).
prop(opposite,upper,1).
prop(opposite,isOrdered,false).
prop(opposite,isUnique,false).
edge(reference,opposite,features).
edge(opposite,reference,owner).
edge(opposite,reference,type).
```

```
/* class Attribute extends StructuralFeature */
node(attribute,class).
prop(attribute,name,"Attribute").
prop(attribute,isAbstract,false).
edge(attribute,structuralFeature,supertypes).

/* datatype Boolean */
node(boolean,dataType).
prop(boolean,name,"Boolean").

/* datatype Integer */
node(integer,dataType).
prop(integer,name,"Integer").

/* datatype String */
node(string,dataType).
prop(string,name,"String").



/*
Helper predicates
*/

is_boolean(true).
is_boolean(false).
is_string(X) :- is_list(X), member(C,X), integer(C).

structuralFeatures(Class,X) :- edge(Class,X,features).
allStructuralFeatures(Class,X) :- structuralFeatures(Class,X).
allStructuralFeatures(Class,X) :- supertypes(Class,SuperClass),
    allStructuralFeatures(SuperClass,X).

conformsTo(Class1,Class2) :- supertypes(Class1,Class2).
conformsTo(Class1,Class2) :- supertypes(Class1,SuperClass), conformsTo(SuperClass,Class2).

isTypeOf(X,Type) :- node(X,Type).

isKindOf(X,Type) :- isTypeOf(X,Type).
isKindOf(X,Type) :- isTypeOf(X,SomeType), conformsTo(SomeType,Type).

supertypes(Class,X) :- edge(Class,X,supertypes).
allSupertypes(Class,X) :- supertypes(Class,X).
allSupertypes(Class,X) :- supertypes(Class,SuperClass), supertypes(SuperClass,X).

subtypes(Class,X) :- supertypes(X,Class).
allSubtypes(Class,X) :- subtypes(Class,X).
allSubtypes(Class,X) :- subtypes(Class,SubClass), allSubtypes(SubClass,X).

oppositeOf(X,Y) :- node(X,reference), node(Y,reference), edge(X,Y,opposite).

lookupElement(Class,ElementName,Element) :- structuralFeatures(Class,Element),
    prop(Element,name,ElementName).
lookupElementExtended(Class,ElementName,Element) :- allStructuralFeatures(Class,Element),
    prop(Element,name,ElementName).

className(Class,Name) :- isTypeOf(Class,class), prop(Class,name,Name).
packageName(Package,Name) :- node(Package,package), prop(Package,name,NameAsList),
    string_to_list(Name,NameAsList).

/*
Some Well-Formedness Rules for KM3-models
*/
hasErrors :- wfr(M,X,Y,Z), writef(M,[X,Y,Z]).
hasErrors :- wfr(M,X,Y), writef(M,[X,Y]).
hasErrors :- wfr(M,X), writef(M,[X]).

wfr('The type of node %w should be a class.',X) :-
```

```
    node(X,Y), not(node(Y,class)).
wfr('Node %w is defined twice with different types %w and %w.',X,Y,Z) :-
    node(X,Y), node(X,Z), \=(Y,Z).
wfr('The type of edge %w->%w should be a reference.',X,Y) :-
    edge(X,Y,Z), not(node(Z, reference)).
wfr('The opposite of reference %w is reference %w, but the reverse is not true.',X,Y) :-
    node(X,reference), edge(X,Y,opposite), not(edge(Y,X,opposite)).
wfr('The opposite of reference %w is reference %w, but types and owner do not match.',X,Y) :-
    node(X,reference), edge(X,XOwner,owner), edge(X,Y,opposite), not(edge(Y,XOwner,type)).
wfr('The target of edge %w->%w is not a valid node.',X,Y) :-
    edge(X,Y,_), not(node(Y,_)).
wfr('The source of edge %w->%w is not a valid node.',X,Y) :-
    edge(X,Y,_), not(node(X,_)).
wfr('The source of edge %w->%w does not have a correct type.',X,Y) :-
    edge(X,Y,Reference), edge(Reference,SourceType,owner), not(isKindOf(X,SourceType)).
wfr('The target of edge %w->%w does not have a correct type.',X,Y) :-
    edge(X,Y,Reference), edge(Reference,TargetType,type), not(isKindOf(Y,TargetType)).
wfr('Edge %w->%w does not have an opposite with appropriate type.',X,Y) :-
    edge(X,Y,Z), oppositeOf(Z,O), not(edge(Y,X,O)).
wfr('%w is an instance of abstract type %w, which is forbidden.',X,Y) :-
    node(X,Y), prop(Y,isAbstract,true).
wfr('Attribute %w is not valid for %w.',X,Y) :-
    prop(Y,X,_), isTypeOf(Y,YType), not(allStructuralFeatures(YType,X)).
wfr('Property %w of %w should be of type Integer.',X,Y) :-
    prop(Y,X,V), edge(X,T,type), node(T,dataType), prop(T,name,"Integer"), not(integer(V)).
wfr('Property %w of %w should be of type String.',X,Y) :-
    prop(Y,X,V), edge(X,T,type), node(T,dataType), prop(T,name,"String"), not(is_string(V)).
wfr('Property %w of %w should be of type Boolean.',X,Y) :-
    prop(Y,X,V), edge(X,T,type), node(T,dataType), prop(T,name,"Boolean"),
    not(is_boolean(V)).
wfr('Classifier %w should be in a package.',X) :-
    isKindOf(X,classifier), not(edge(X,_,me_package)).
wfr('Element %w should have a value for attribute %w.',X,Y) :-
    isTypeOf(X,XType), allStructuralFeatures(XType,Y), node(Y,attribute),
    not(prop(Y,lower,0)), not(prop(X,Y,_)).
wfr('Element %w should have a value for reference %w.',X,Y) :-
    isTypeOf(X,XType), allStructuralFeatures(XType,Y), node(Y,reference),
    not(prop(Y,lower,0)), not(edge(X,_,Y)).
wfr('Element %w is contained in both %w and %w.',X,Y,Z) :-
    edge(X,Y,F1), edge(X,Z,F2), oppositeOf(F1,OF1), oppositeOf(F2,OF2),
    prop(OF1,isContainer,true), prop(OF2,isContainer,true), \=(F1,F2).
```