The Level-Agnostic Modeling Language

Colin Atkinson, Bastian Kennel, and Björn Goß

Chair of Software Technology, University Mannheim
A5, 6 B 68131 Mannheim, Germany
{Colin.Atkinson,Bastian.Kennel,
Bjoern.Goss}@informatik.uni-mannheim.de

Abstract. As an alternative modeling infrastructure and paradigm, multi-level modeling addresses many of the conceptual weaknesses found in the four level modeling infrastructure that underpins traditional modeling approaches like UML and EMF. It does this by explicitly distinguishing between linguistic and ontological forms of classification and by allowing the influence of classifiers to extend over more than one level of instantiation. Multi-level modeling is consequently starting to receive attention from a growing number of research groups. However, there has never been a concrete definition of a language designed from the ground-up for the specific purpose of representing multi-level models. Some authors have informally defined the "look and feel" of such a language, but to date there has been no systematic or fully elaborated definition of its concrete syntax. In this paper we address this problem by introducing the key elements of a language, known as the Level-Agnostic Modeling Language (LML) designed to support multi-level modeling.

Keywords: Multi-Level Modeling, Modeling Language.

1 Introduction

Since it kick started the modern era of model driven development in the early 90s, the concrete syntax of the structural modeling features of the UML has changed very little. Almost all the major enhancements in the UML have focused on its abstract syntax and infrastructure [1]. Arguably, however, the original success of the UML was due to its concrete syntax rather than its abstract syntax as the former has been adopted in many other information modeling approaches, such as ontology and data modeling [9]. Today, discussions about the concrete syntax of languages usually take place in the context of domain specific modeling languages [3]. Indeed, domain specific languages are increasingly seen as one of the key enabling technologies in software engineering. While the importance of domain specific languages is beyond doubt, universal languages that capture information in a domain independent way still have an important role to play. They not only provide domain-spanning representations of information, they can tie different domain specific languages together within a single framework.

Existing universal languages such as the UML are not set up to fulfill this role. The UML's infrastructure not only has numerous conceptual weaknesses [5], it does not

B. Malloy, S. Staab, and M. van den Brand (Eds.): SLE 2010, LNCS 6563, pp. 266–275, 2011.
© Springer-Verlag Berlin Heidelberg 2011

satisfactorily accommodate the newer modeling paradigms that have become popular in recent years, such as ontology engineering [2]. In this paper we present a proposal for a new universal modeling language that addresses these weaknesses. Known as the Level-agnostic Modeling language, or LML for short, the language is intended to provide support for multi-level modeling, seen by a growing number of researchers as the best conceptual foundation for class/object oriented modeling [6], [7], [8]. It is also intended to accommodate other major knowledge/information modeling approaches and to dovetail seamlessly with domain specific languages. Since the core ideas behind multi-level modeling have been described in a number of other publications, in this paper we focus on the concrete syntax of the LML and only provide enough information about the semantics and abstract syntax to make it understandable. The language is essentially a consolidation and extension of the notation informally employed by Atkinson and Kühne in their various papers on multi-level modeling [6], [11], [12]. Some of the concrete goals the LML was designed to fulfill are as follows:

To be UML-like. Despite its weaknesses, the UML's core structural modeling features, and the concrete syntax used to represent them, have been a phenomenal success and have become the de facto standard for the graphical representation of models in software engineering. To the greatest extent possible the LML was designed to adhere to the concrete syntax and modeling conventions of the UML.

To be level agnostic. The language was designed to support multi-level modeling based on the orthogonal classification architecture described by Atkinson, Kühne and others [6], [7], [12]. The core requirement arising from this goal is the uniform (i.e. level agnostic) representation of model elements across all ontological modeling levels (models) in a multi-level model (ontology).

To accommodate all mainstream modeling paradigms. Although UML is the most widely used modeling language in software engineering, in other communities other languages are more prominent. For example, in the semantic web and artificial intelligence communities ontology languages like OWL [9] are widely used, while in the database design community Entity Relationship modeling languages such as those proposed by Chen [13] are still important. A key goal of the LML therefore is to support as many of these modeling paradigms as possible, consistent with the overall goal of being UML-oriented.

To provide simple support for reasoning services. The lack of reasoning services of the kind offered by languages such as OWL is one of the main perceived weaknesses of UML. However, there are many forms and varieties of reasoning and checking services provided by different tools, often with non-obvious semantics. One goal of the LML is therefore to provide a foundation for a unified and simplified set of reasoning and model checking services.

Unfortunately, due to space restrictions it is only possible to provide a general overview of LML's features, and it is not possible to discuss LML's support for reasoning services in any detail. In the next section we provide a brief overview of multi-level modeling and the infrastructure that underpins the LML. In section 3 we then present the main features of the language in the context of a small case study motivated by the movie Antz. In section 4 we discuss logical relationships, which are one of the

foundations for the reasoning services built on the LML, before concluding with some final remarks in section 5.

2 Orthogonal Classification Architecture

As illustrated in Fig. 1, the essential difference between the orthogonal classification architecture and the traditional four level modeling architecture of the OMG and EMF is that there are two classification dimensions rather than one. In the linguistic dimension, each element in the domain modeling level (L1) is classified according to its linguistic type in the L0 level above. Thus, elements such as *AntType*, *Ant*, *Queen* and the relationships between them are defined as instances of model elements in the L0 level according to their linguistic role or form. In the ontological dimension, each element in an ontological level (e.g. O1, O2, ..) is classified according to its ontological type in the ontological level above (i.e. to the left in Fig. 1). Thus *Bala* is an ontological instance of *Queen,* which in turn is an ontological instance of *AntType*. The elements in the top level in each dimension (with the label 0) do not have a classifier. The bottom level in Fig. 1 is not part of the modeling infrastructure, per se, but contains the entities that are represented by the model element in L1. Like the bottom level of the OMG and EMF modeling infrastructures, therefore, it can be regarded as representing the "real world".

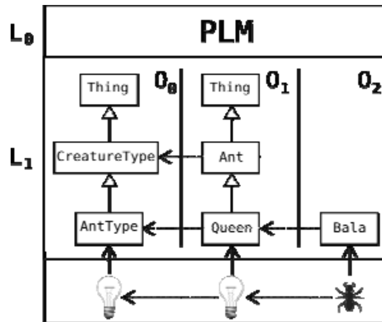

Fig. 1. Multi-Level Modeling Example

In principle, it is possible to have an arbitrary number of linguistic levels, but in practice the arrangement shown in Fig. 1 is sufficient. In our orthogonal classification architecture there are only two linguistic levels and only one of these, L1, is divided into ontological levels. The top level linguistic model therefore defines the underlying representation format for all domain-oriented and user-defined model content in the L1 level, and spans all the ontological levels in L1. For this reason we refer to it as the Pan-Level Model or PLM for short. In effect, it plays the same role as the MOF and Ecore in the OMG and EMF modeling infrastructures, but its relationship to user models respects the tenet of strict meta modeling [5].

The Pan Level Model (PLM) defines the abstract syntax of the language that we introduce in the ensuing sections [10]. The LML therefore essentially defines a concrete

syntax for the PLM. To simplify discussions about content organized in the way shown in Fig. 1 we use the terms “ontology” and “model” in a specific way. We call all the modeling content in the L1 level, across all ontological levels, an “ontology”, while we call the modeling content within a single ontological level a “model”. This is consistent with common use of the terms since models in the UML are usually type models [14] and ontologies often define the relationships between instances and types (i.e. the relationship between two ontological levels or models).

There are two basic kinds of elements in an LML ontology: *InstantiatableElements* and *LogicalElements,* each of which has a *name* attribute and an *owner* attribute. *InstantiatableElements* are the core modeling elements that fulfill the role of classes, objects, associations, links and features in traditional modeling languages such as the UML, OWL or ER languages. The main difference is that they fulfill these roles in a level-agnostic way. *LogicalElements* on the other hand represent classification, generalization and set theoretic relationships between *InstanstiatableElements*. *InstantiatableElements* come in two basic forms, *Clabjects*, which represent classes, objects, associations and links, and *Features* that play a similar role to features in the UML – that is, attributes, slots and methods. *Field* is the subclass of *Feature* that plays the role of attributes and/or slots, while *Method* is the subclass of *Feature* that plays the role of methods. *Clabjects*, in general, can simultaneously be classes and objects (hence the name – a derivative of “class” and “object”). In other words they represent model elements that have an instance facet and a type facet at the same time. There are two kinds of *Clabjects*, *DomainEntities* and *DomainConnections*. *DomainEntities* are *Clabjects* that represent core domain concepts, while *DomainConnections*, as there name implies, represent relationship between *Clabjects*. They play the role of associations and links, but in a way that resembles association classes with their own attributes and slots.

3 Clabjects

The concrete representation of clabjects in LML is based on the notational conventions developed by Atkinson and Kühne [11] which in turn are based on the UML. Fig. 2 shows an example multi-level model (i.e. ontology) rendered using the LML concrete syntax. Inspired by the film Antz, this contains three ontological levels (i.e. models) organized vertically rather than horizontally as in Fig. 1.

In its full form a clabject has three compartments: a header compartment, a field compartments and a method compartment. The header is the only mandatory compartment and must contain an identifier for the clabject, shown in bold font. This can be followed, optionally, by a superscript indicating the clabject's potency and a subscript indicating the ontological level (or model) that it occupies. Like a multiplicity constraint, a potency is a non-negative integer value or “*” standing for “unlimited” (i.e. no constraint). Thus the clabject *FlyingAnt*, in Fig. 2 has potency 1 and level 1, while clabject *Z* has potency 0 and level 2. The field and method compartments are both optional. They intentionally resemble the attribute and slot compartments in UML classes and objects. However, the key difference once again is that the notation for fields is level agnostic. This allows them to represent UML-like attributes, slots and a mixture of both, depending on their potency.

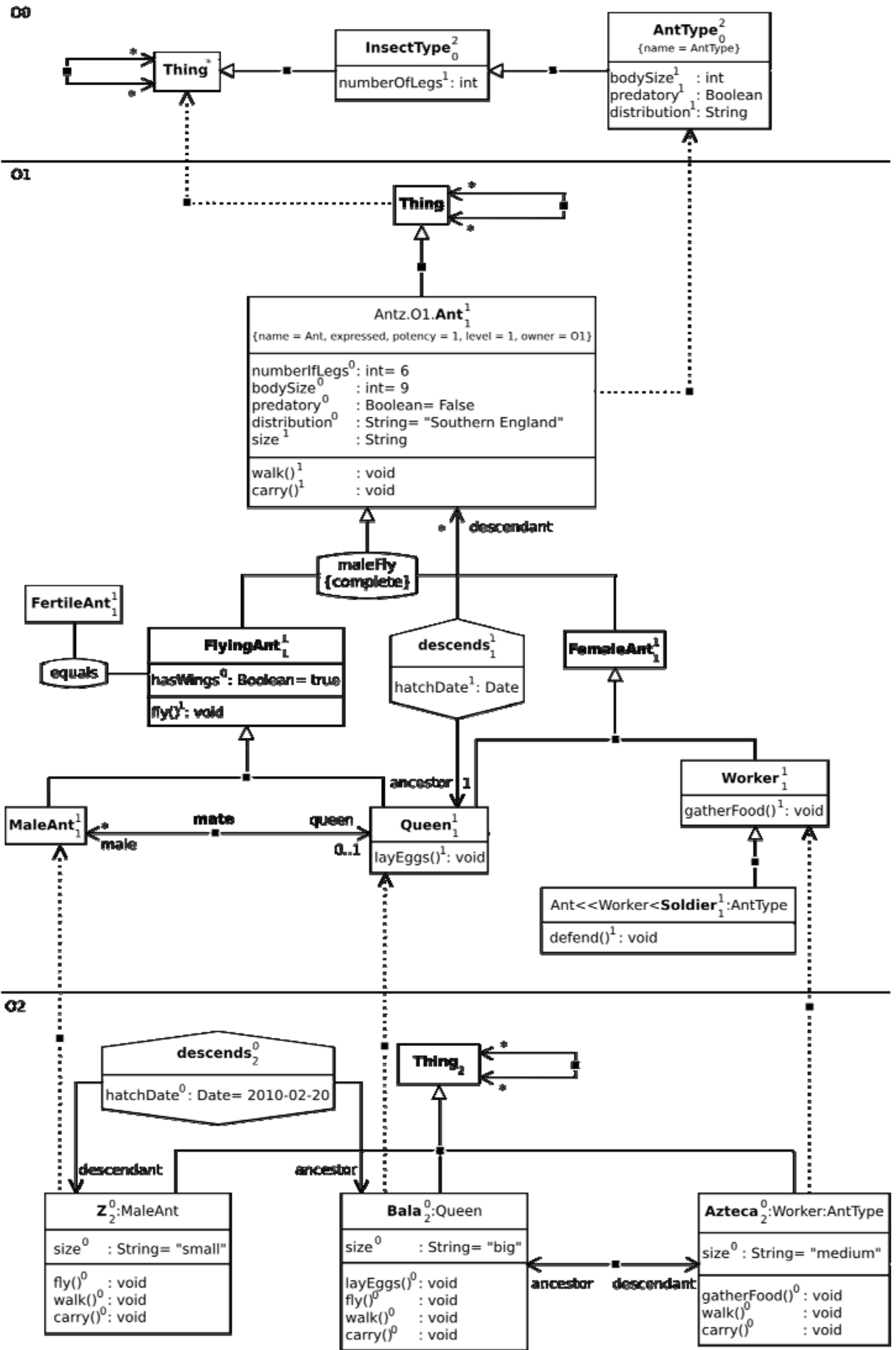

Fig. 2. Antz domain example in LML concrete syntax

Potencies on fields follow the same rules as potencies on domain entities. The only constraint is that a field cannot have a higher potency than the clabject that owns it. In general, the representation of a field has three parts: a name, which is mandatory, a type and a value. A traditional UML slot corresponds to a potency 0 field with a value, whereas a traditional UML attribute corresponds to a potency 1 field without a value.

When a value is assigned to a field with a potency greater than 0, the value represents the default value for corresponding fields of the clabject's instances. In Fig. 2, *FlyingAnt* has a potency 0 field, *hasWings,* of type *Boolean* with the value T*rue*.

Methods are represented in the same way as operations in the UML using signature expressions that include the name, the input parameters and the return value if there is one. The only difference is that methods in LML have a potency associated with them. As with fields, the potency of a method cannot be higher than the potency of its owning clabject.

3.1 Proximity Indication

The UML provides various notational enhancements in the header compartment of classes and objects to show their location within the instantiation, inheritance and containment hierarchies. More specifically, the ":" symbol can be used to to express the fact that an object is an instance of a class and the "::" notation can be used to show that a model element belongs to, or is contained in, another model element. However, these notations are rather ad hoc and cannot be used in a level agnostic way. For example, to show that a class is an instance of another class at a higher meta-level, a stereotype has to be "attached" to the class using the guillemot notation. Alternatively, the powertype concept can also be used to indicate that a class X is an instance of another class Y, but only if it is one of numerous subclasses of another class, Z. In contrast to the UML, the LML provides a fully level-agnostic way of representing a clabject's location in the instantiation, inheritance and containment hierarchies.

Classification (Instantiation) Hierarchy. The basic notation for representing a clabject's location in the instantiation hierarchy in the LML is the same as that used in the UML for objects. In Fig. 2, the header of the *Worker* clabject, "Worker:AntType", indicates that *Worker* is an instance of *AntType*. Of course this notation can also be applied over multiple ontological levels as shown in the header of *Azteca* ("Azteca:Worker:AntType"). It is also possible to use the "::" notation to omit one one or more clabjects in the instantiation hierarchy. For example, if it is only of relevance that *Azteca* is an instance of *AntType*, *Azteca*'s header can be reduced to "Azteca::AntType".

Generalization (Inheritance) Hierarchy. The UML provides no way of showing the ancestry of a class within its header compartment. In LML this can be done using the "<" symbol on the left hand side of a clabject's name to represent subtypeOf relationships. This is intended to resemble the white triangle at the supertype end of a generalization in the UML. As with the "::" notation, two consecutive "<" characters can be used to indicate that one or more clabjects of the inheritance hierarchy are not identified. Thus, the header of *Soldier* in Fig. 2 shows that it is a subclass of *Worker* and *Ant*, but omits the fact that it is also a subclass of *FemaleAnt*. Of course, in Fig. 2 this

information is redundant in the header of *Soldier* because the inheritance hierarchy is shown explicitly using generalization relationships.

Ownership (Containment) Hierarchy. Ownership is the basis for defining namespaces. Basically, an element is a namespace for everything that it owns - that is, anything that identifies it as its owner through its owner attribute. The relationship between a clabject and its *Fields* is also ownership. *Fields* have the clabject they belong to as their owner. Ownership information can be included in the identifier of a clabject, like the *Ant* clabject in Fig. 2. In LML this is shown using the "." symbol rather than the "::" symbol as in the UML. This is consistent with the notation used in programming languages such as Java. Thus, the header of *Ant* in Fig. 2, ""Antz.O1.Ant", shows that *Ant* is contained in the model *O1* which in turn is contained in the ontology *Antz*.

3.2 Attribute Value Specifications

In traditional modeling environments, model elements only derive their attributes from one type, and in UML like languages these are shown in the compartment below the header. In a multi-level modeling environment, all model elements except those at the top ontological level (i.e. model) have two types, a linguistic one and an ontological one. The attributes of the ontological types are shown in the field compartment in the traditional way. The linguistic attributes, in contrast, are shown elsewhere in one of two different ways. The first way uses special notations or conventions for different attributes, such as the subscript and superscript notations for level and potency. The second way is to add explicit "attribute specifications" under the name of a clabject, similar to tagged value specifications in the UML. The general form of an attribute value specification, is the following

```
{attributeName = value, attributeName = value, ...}
```

For boolean attributes the UML convention applies. If a boolean attribute is true it is only necessary to include its name in the list. However, if this convention is used then all true boolean attributes of the clabject must be shown as well. This is because all boolean attributes that are not shown are assumed to be false. In Fig. 2 the *Ant* clabject has an attribute value specification indicating the value of it attributes.

3.3 Domain Connections

As mentioned in section 2, there are two distinct kinds of clabjects – domain entities and domain connections. The former play a role similar to classes/objects in the UML and the latter play a role similar to associations/links. As shown in Fig. 2 they are distinguished visually by different symbols - rectangles in the case of domain entities and flattened hexagons in the case of domain connections. This is intended to resemble the diamond symbol that is used to represent connections in Entity Relationship diagrams [13]. ERA diagrams can therefore easily be represented in LML. We use flattened hexagons rather than diamonds because they are ergonomically more compact.

The other major difference is that domain connections are responsible for "carrying" the lines that are used in the representation of connections. The problem with

using a symbol like a hexagon to represent a domain connection is that this breaks the fundamental visualization metaphor of the UML in which relationships are mapped to edges rather than nodes. To address this problem and accommodate both the UML and ERA visualization metaphors at the same time, LML allows a domain connection to be represented in an imploded form by a "visually insignificant" dot as well as in the full exploded form as a flattened hexagon [15]. In the former case, the domain connection is still conceptually represented as a node, but the overall visual effect is that of an edge as in the UML. In Fig. 2, the domain connection *mate* is shown in this dotted form. This shows that the multiplicity constraints, role names and navigability values owned by the domain connection can be shown at the end of the appropriate lines using the standard UML conventions. In general, it is possible to present all the information that can appear in the header of the expanded form of a domain connection next to the dot. This includes the name and an attribute value specification. However, the fields and methods of a domain connection can only be shown in the expanded form, as indicated by the *descends* connections in Fig. 2.

4 Logical Elements

Logical relationships capture set theoretic relationships between clabjects and come in three basic forms – *Generalization*, *Instantiation*, and *SetRelationship*. The first kind captures subtyping/supertyping relationships between clabjects, the second kind captures classification relationships between clabjects and the third kind captures other general set theoretic relationships between clabjects.
Since they are relationships, the same basic representation options used for *DomainConnections* is also supported for *LogicalElement*s – namely, they can be rendered in an exploded form and in an imploded form as a "dot". The expanded form uses a slightly different symbol to *DomainConnections* – namely, a "rectangle" with rounded sides at the top and bottom. Another minor difference is that for logical relationships the name is optional in the expanded form.

Generalization. Generalization relationships show the basic subtype/supertype relationships in set-based models. In Fig. 2, the connection-line between *Ant* and *Thing* using the white triangular arrow (which is similar to the UML representation of generalizations) indicates that *Ant* is a subtype of *Thing* in the imploded representation. The generalization, *maleFly,* on the other hand shows generalization in an exploded form. It indicates that *FlyingAnt* and *FemaleAnt* are both subtypes of *Ant*. Besides the generalization's name the generalization symbol contains an attribute value specification indicating that *maleFly* is complete.

Instantiation. Instantiation relationships show that one clabject is an instance of another clabject. As such they are the only kind of relationships that are allowed to cross an ontological level boundary (in fact they have to) to connect two clabjects at different levels. As the instantiation between *Bala* and *Queen* in Fig. 2 shows, *Instantiation* relationships are represented in a similar way to the UML using an open-headed, dashed arrow going from the instance to the type. They can also be represented in an imploded or exploded form.

SetRelationship. Set relationships specificity some other kind of set theoretic relationship between clabject. Typical examples are *complements, inverseOf* and

equals. The equals relationship between *FlyingAnt* and *FertileAnt* in Fig. 2 indicates that they essentially represent the same set, since every fertile ant can fly and only fertile ants can fly. As usual the name of the relationship is shown inside the shape. Set relationships do not normally have attribute value specifications.

5 Conclusion

In this paper we have presented a language, LML, designed to support multi-level modeling using the orthogonal classification architecture, with a focus on its concrete syntax. As mentioned in the introduction this work builds on the informal notation developed by Atkinson and Kühne in a series of papers. As well as supporting multi-level modeling the language has also been designed to exhibit several other important characteristics. However, for space reasons it is not possible to fully explain them all in a paper of this size.

First, to the greatest extent possible, LML was designed to support and be consistent with the notations and conventions popularized by the UML. As is hopefully evident from the discussion, LML can easily be made to have the look and feel of traditional UML. Like the UML, the LML concrete syntax is intended to be renderable in black and white and to be readily drawable by hand where necessary. There are lots of ways in which colour could enhance the information shown in LML diagrams, but this is left to individual tools and modelers.

Second, LML is also designed to support the look and feel of other important modeling paradigms as well. In particular it can support the entity relationship modeling approach pioneered by Chen for data modeling and contains all the features needed to represent OWL ontologies. Because it supports all ontological levels in a uniform and relatively rich way, the LML provides a natural representation of both instance and type knowledge. OWL, in contrast, only directly supports the latter.

Third, the language has built-in support for depicting which information in a model has been explicitly expressed by a human modeler and which has been computed by a reasoning or transformation engine. This allows the LML to support simpler, more understandable reasoning services. For example the traditional "subsumption" service offered by ontology engineering tools such as Protegé [16] essentially computes new generalization and classification relationships based on the properties of the classes and objects in the ontology. With LML, these could be offered in a more understandable way through services such as "add all generalizations" or "add all instantiation relationships". This not only breaks down the service into its constituent parts which can be invoked independently, it also allows them to be fine tuned into more useful services.

Finally, the language includes various kinds of elision symbols which can show when expressed information is not shown in a particular view of a model. In other words, it can be used to show that the underlying model or knowledge base has further expressed information that is not shown in the current diagram. This is important in clarifying whether a diagram, and queries direct to it, should be interpreted in an "open world" or a "close world" way.

Although the focus of modeling language research has turned from universal modeling languages like the UML to DSLs we believe the evolution of universal modeling

languages is far from over. We hope the ideas presented in this paper will contribute towards this evolution and will help lead towards a better synergy of DSL and universal modeling languages. We also hope it will pave the way for a unification of software modeling (i.e. UML-oriented), knowledge representation (i.e. OWL-oriented) and information modeling (i.e. ER-oriented) technologies.

References

1. OMG UML 2.1.2 Infrastructure Specification, Object Management Group (OMG), Tech. Rep. (2007)
2. Gasevic, D., Djuric, D., Devedzic, V., Damjanovi, V.: Converting UML to OWL ontologies. In: Proceedings of the 13th International World Wide Web, New York, NY (2004)
3. Tolvanen, J.: MetaEdit+: Domain-Specific Modeling for Full Code Generation Demonstrated. In: Proc. 19th Ann. ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications (2004)
4. GreenField, J., Short, K., Cook, S., Kent, S., Crupi, J.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley and Sons, Chichester (2004)
5. Atkinson, C., Kühne, T.: Rearchitecting the UML Infrastructure. ACM Journal Transactions on Modeling and Computer Simulation 12(4) (2002)
6. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software (2003)
7. Asikainen, T., Männistö, T.: Nivel:a metamodelling language with a formal semantics. Software and Systems Modeling 8(4), 521–549 (2009)
8. Aschauer, T., Dauenhauer, G., Pree, W.: Multi-level Modeling for Industrial Automation Systems. In: Software Engineering and Advanced Applications, Euromicro Conference, pp. 490–496 (2009)
9. OWL (2004), http://www.w3.org/2004/OWL
10. Atkinson, C., Gutheil, M., Kennel, B.: A Flexible Infrastructure for Multi-Level Language Engineering. IEEE Transactions on Software Engineering 35(6) (2009)
11. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 19–33. Springer, Heidelberg (2001)
12. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. In: Software and Systems Modeling (2007)
13. Chen, P.: The Entity-Relationship Model: Toward a Unified View of Data. ACM Transactions on Database Systems 1, 9–36 (1976)
14. Kühne, T.: Matters of (Meta-)Modeling. Journal on Software and Systems Modeling 5(4), 369–385 (2006)
15. Gutheil, M., Kennel, B., Atkinson, C.: A Systematic Approach to Connectors in a Multi-Level Modeling Environment. In: Proc. 11th Int'l Conf. Model Driven Eng. Languages and Systems (2008)
16. Protege Tool (2008), http://protege.stanford.edu