A Flexible Infrastructure for Multilevel Language Engineering

Colin Atkinson, *Member*, *IEEE*, Matthias Gutheil, and Bastian Kennel

Abstract—Although domain-specific modeling tools have come a long way since the modern era of model-driven development started in the early 1990s and now offer an impressive range of features, there is still significant room for enhancing the flexibility they offer to end users and for combining the advantages of domain-specific and general-purpose languages. To do this, however, it is necessary to enhance the way in which the current generation of tools view metamodeling and support the representation of the multiple, "ontological" classification levels that often exist in subject domains. State-of-the-art tools essentially allow users to describe the abstract and concrete syntaxes of a language in the form of metamodels and to make statements in that language in the form of models. These statements typically convey information in terms of types and instances in the domain (e.g., the classes and objects of UML), but not in terms of types of types (i.e., domain metaclasses), and types of types of types, and so on, across multiple classification levels. In essence, therefore, while they provide rich support for "linguistic" metamodeling, the current generation of tools provides little if any built-in support for modeling "ontological" classification across more than one type/instance level in the subject domain. In this paper, we describe a prototype implementation of a new kind of modeling infrastructure that, by providing built-in support for multiple ontological as well as linguistic classification levels, offers various advantages over existing language engineering approaches and tools. These include the ability to view a single model from the perspective of both a general-purpose and a domain-specific modeling language, the ability to define constraints across multiple ontological classification levels, and the ability to tie the rendering of model elements to ontological as well as linguistic types over multiple classification levels. After first outlining the key conceptual ingredients of this new infrastructure and presenting the main elements of our current realization, we show these benefits through two small examples.

Index Terms—Language engineering, metamodeling, multilevel modeling.

1 Introduction

The need to provide users with the ability to engineer new languages was already evident when UML was first designed in the mid 1990s, kickstarting the modern era of model-driven development. Although UML was primarily designed to be a general-purpose modeling language, even the first version had a rudimentary "lightweight" customization mechanism that enabled model elements to be "branded" using "stereotypes" and, over time, this evolved into the rich profile mechanism supported in the current version of the language [1], [2].

Today, language engineering is an established segment of the modeling tool market and a well-recognized research domain with its own community and conferences such as the International Software Language Engineering conferences [3] or the OOPSLA Domain-Specific Modeling workshops [4]. All DSL engineering tools basically make it possible for users to define the abstract and concrete syntaxes of a language as a model, and to make statements in that language through other models (the latter being regarded as instances of the former). Some tools provide this capability around UML (e.g., GMF [5]), while others provide approaches that are fully independent of UML (e.g., Microsoft Software Factories [6], MetaEdit+ [7], and GME [8]).

The successful use of such Domain-Specific Modeling (DSM) tools to enhance specific phases of software engineering projects—for example, to accelerate the requirements engineering process—demonstrates that they already provide significant value. However, the fact that they are usually restricted to niche roles and are rarely used to carry the main development load also indicates that they are still immature and there is room to improve their capabilities and usability.

We believe that one significant weakness shared by all leading DSM tools today is their asymmetric treatment of ontological and linguistic classification when extended over multiple levels. Linguistic classification—the basic type/instance relationship that exists between an element of a model and an element of the abstract syntax of the language used to express the model—is not only used to define the basic modeling architecture of current DSM tools, it is also the basis for their constraint checking mechanisms and their model rendering algorithms. For example, in UML-oriented tools, the M_2 level (the "metalevel" where domain-specific languages are defined) contains the linguistic classifiers (abstract and concrete syntax elements) of elements of models at the M_1 level, and is used to determine how they are rendered and whether they are well formed. Similarly, the M_3 level (the "metameta level" where the language for defining domain-specific languages is defined) contains the

- *C. Atkinson and B. Kennel are with the Institute for Computer Science, University of Mannheim, A 5, 6 Bebäudeteil B, Seminargebäde, D-68131 Mannheim, Germany. E-mail: {atkinson, bastian.kennel}@informatik.uni-mannheim.de.*
- *M. Gutheil is with itemis AG, Augustusring 32, 53111 Bonn, Germany. E-mail: matthias.gutheil@itemis.de.*

Manuscript received 4 May 2008; revised 9 Nov. 2008; accepted 18 Jan. 2009; published online 23 Apr. 2009.
Recommended for acceptance by J.-M. Favre, D. Gašević, R. Lämmel, and A. Winter.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2008-05-0169. Digital Object Identifier no. 10.1109/TSE.2009.31.

linguistic classifiers (abstract and concrete syntax elements) of elements of Domain-Specific Language (DSL) descriptions at the M_2 level, and is used to determine how the description is rendered and whether it is well formed. Tools that are not focused on the UML such as MetaEdit+ [7] and GME [8] also have the same basic architecture, although their modeling levels are not explicitly related to the UML infrastructure levels.

The limitations of this reliance on linguistic classification to drive DSM tools become evident when one wishes to model, in a domain-specific way, domain concepts that do not fit into a single type/instance level. Consider the domain concepts "BillGates," "ITProfessional," and "Profession," for example. While "BillGates" is clearly an instance of "ITProfessional" (i.e., the latter is the classifier of the former), he is clearly not an instance of "Profession" ("BillGates" is not a "Profession")—it is "ITProfessional" that is an instance of "Profession" ("ITProfessional" is a "Profession"). Situations like this, where the named concepts in a domain naturally span three or more classification levels, are common in many domains. However, they are not directly supported in current DSM tools because the ontological classification relationship upon which they are based is either not recognized as a classification relationship by the tool, or is only supported for one type/instance dichotomy (i.e., instance and types but not ontological metatypes and beyond, etc.). Modelers must therefore use some kind of "work around" in which the domain concepts are all modeled as classes and the classification relationships between them are simulated by associations [9]. When using such a work around, the logical classifiers can have no role in determining whether their instances are well formed or how they are rendered. In short, their role as classifiers is ignored by current DSM tools, which focus exclusively on linguistic classification.

During the period from 1997 to 2005, Atkinson and Kühne [10], [11] and other authors [12], [13], [14] laid down some of the basic principles and concepts needed to support a more symmetric modeling infrastructure that treats linguistic and ontological classification in a more equal way and gives them both a role in controlling the structuring and rendering of models. The overall framework is referred to as the orthogonal classification architecture (OCA) by Atkinson and Kühne since it essentially applies the conventional UML metamodeling hierarchy in two orthogonal dimensions [10]. However, although tools have been built that exhibit some of the identified characteristics, to our knowledge, no language engineering tool has yet been built that embraces the principles of the OCA in a fundamental way from the ground up. Kühne and Schreiber have built a tool that applies orthogonal classification for the purpose of coding [15], but not for language engineering.

The advantages to the end user of such an architecture are twofold. First, when multiple ontological classification levels exist in a domain, users receive more direct support for modeling them and are able to capture related constraints using normal classification-oriented features of constraint languages like the Object Constraint Language (OCL) [16]. Second, because both the ontological and classification dimensions can be used to drive the rendering process, models can have two representations at the same time—one, a highly domain-specific representation driven by ontological classifiers and the other a more general representation driven by the linguistic classifiers. Having the ability to view a model in both a domain-specific and general way combines the advantages of domain-specific and general-purpose modeling languages, and alleviates one of the main problems of domain-specific languages—the so-called "Tower of Babel" problem in which the proliferation of different languages reduces the value of models as communication vehicles.

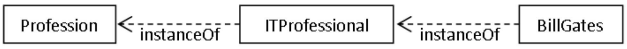

Fig. 1. Type hierarchy.

In this paper, we describe work that we have performed in building a prototype language engineering environment based on the OCA architecture. In the next section, we review the key features of the OCA as outlined by Atkinson and Kühne and add some new requirements and solutions that we developed during the course of this work. Then, in Section 3, we outline the details of the framework and give two small examples of its use in Section 4. Section 5 describes related work, and Section 6 discusses the potential significance of the approach. Finally, in Section 7, we conclude.

2 Language Engineering Issues

In order to explain and motivate our approach, in this section, we identify the main issues involved in providing more balanced support for ontological and linguistic classification in DSM tools. Each of the following sections has four parts: The "**issue**" part describes a particular issue and why it is important, the "**current status**" part describes to what extent this issue is addressed in existing technologies/tools and how, the "**our approach**" part discusses our strategy for addressing the issue, and the "**other approaches**" part discusses other suggested and potential solutions to the problem. Our suggested solution to each issue effectively constitutes a requirement that we believe an ideal tool for DSM should support. In Section 3, we describe how our prototype tool meets these requirements.

2.1 Dual Classification

Issue. The dual classification issue arises when a language whose abstract syntax is defined using the notion of types and instances (i.e., elements of statements made in the language are instances of abstract syntax types) also supports the explicit representation of type/instance relationships in statements made in the language.

Consider Fig. 1, for example. This is a statement made in a UML-like modeling language that describes domain concepts spanning three instantiation levels—it contains an instance, a type, and a metatype. The intended meaning of this statement is that *BillGates* is an instance of *ITProfessional* and that *ITProfessional*, in turn, is an instance of *Profession*. If the language used to make this statement is itself defined using the same basic concepts of classes and instances, the elements appearing in statements of the language are also instances of the metamodel elements used to define the language, as illustrated in Fig. 2.

Most of the model elements in this diagram are instances of two types—they are an instance of an abstract syntax element (their linguistic type) and a "logical" instance of another model element in the diagram (their ontological

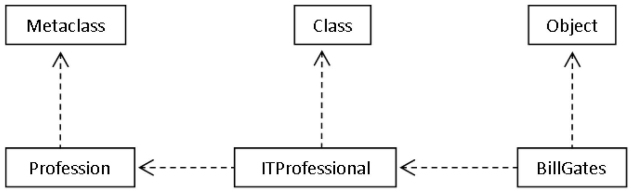

Fig. 2. Dual classification.

type). The latter, depicted by horizontal arrows in Fig. 2, represent ontological relationships in the subject of the model (i.e., the domain being modeled), while the former (represented by vertical arrows in Fig. 2) defines the linguistic role of each model element.

Current status. The current generation of modeling tools almost exclusively focuses on the linguistic classification dimension and provides little if any direct support for modeling ontological classification relationships across more than two levels. They can support the creation of visual diagrams such as that depicted in Fig. 2, but the *instanceOf* relationship in such diagrams has no special meaning to the tool. For example, tools such as GME [8] or MetaEdit+ [7] would regard the model in Fig. 1 as a statement (at level M_1) in a domain-specific language defined as a metamodel at M_2. While the visual depiction of such a model is no problem for such tools, their constraint checking and visualization mechanisms are exclusively focused on linguistic classification. Thus, although they have built-in support for checking that constraints defined in the linguistic metamodel are satisfied by a statement expressed in the language (i.e., in a model), they have no built-in support for checking that a constraint applied to an M_1 class is adhered to by its ontological instances at the same linguistic level.[1] Similarly, they provide flexible mechanisms for rendering model elements in different ways based on their linguistic classifiers, but not on their ontological classifiers. In effect, the tools are "blind" to ontological classification when it comes to providing support for domain-specific rendering and constraint checking.

The situation is similar for UML-based tools. In the early days of the UML when only two levels were considered, types (e.g., *ITProfessional*) were thought of as occupying M_1, and instances (e.g., *BillGates*) were thought of as occupying M_0. *MetaClasses* such as "Profession," if supported at all, would have been regarded as occupying M_2. However, UML2.0 changed this interpretation of the levels by regarding M_0 as containing "real-world" elements and M_2 as being the language definition (i.e., the location for linguistic classifiers making up the abstract syntax). Thus, in the new version of the UML infrastructure, the model elements in Fig. 1 are all regarded as occupying the M_1 level of the model hierarchy, and each logical level is given its own linguistic classifier. The abstract syntax element *InstanceSpecification* is the linguistic classifier for instances (e.g., *BillGates*), *Classifier* is the general linguistic classifier for classes (e.g., *ITProfessional*) and *MetaClass* is a stereotype than can be used to label logical metaclasses.

1. A modeler can "work around" this to some extent by writing constraints that treat *instanceOf* relationships as associations (i.e., regular connectors), but the built-in operations relying on type/instance semantics (e.g., *allInstances, isTypeOf*, etc.) are not supported.

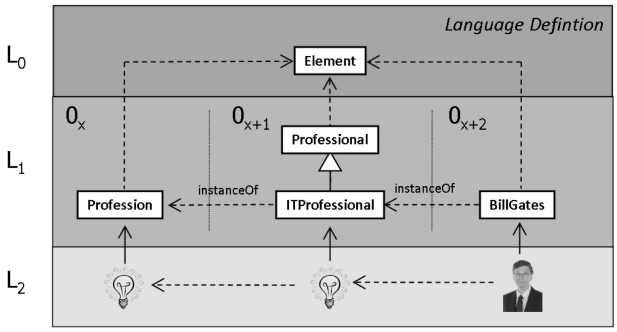

Fig. 3. Orthogonal classification architecture.

This approach not only treats the model elements at different ontological levels in a nonuniform way, it limits the number of ontological levels that can be modeled to the number of distinct linguistic classifiers provided to identify the different levels. Furthermore, like the non-UML tools, built-in constraint definition and checking features are built around linguistic classification, as are the visualization mechanisms. Thus, the classification-related features of OCL, such as *allInstances* and *isTypeOf*, etc., do not work on ontological classification.

Our solution. To address this issue, we believe that a language engineering environment should be built from the ground up to support both forms of classification and should provide built-in languages and mechanisms that support constraint checking and element rendering driven by ontological as well as linguistic classification. In other words, we believe that ontological classification should be given equal weight to linguistic classification.

Since it is based on two fundamentally orthogonal classification dimensions, as illustrated in Fig. 3, we refer to this architecture as the OCA [11]. This shows that there are two separate dimensions of classification—linguistic instantiation (going vertically in Fig. 3) and ontological classification (going horizontally in Fig. 3). It also shows that each dimension can be divided into levels according to the principles of strict metamodeling[2] [11], and that each element in the user-defined model (L_1) is classified in two ways (except the elements in the left most ontological level).

The diagram also shows that all model elements are instances of the same linguistic types—that is, there are no special linguistic elements to represent model elements belonging to a specific ontological level (e.g., *InstanceSpecification*, *Classifier*, *MetaClass*, etc.) since this obviously places an artificial limit on the number of ontological levels that can exist.

Other solutions. The de facto existence of two distinct forms of classification in a multilevel modeling frameworks was recognized over a decade ago by Geisler et al. [18] and later by Bezevan and Gerbe [19], although these authors did not use the terms "linguistic classification" and "ontological classification." Nor did they attempt to rearchitect traditional modeling frameworks to leverage them. The first attempt to do so was reported in 2001 by Riehle et al. [20]. More recently, Gonzalez-Perez and Henderson-Sellers [21]

2. This principle holds that only *instanceOf* (classification) relationships can cross model-level boundaries, and that every element in one level must be a direct instance of exactly one element of the level above (except the top level where a special solution is needed to terminate the model-level hierarchy).

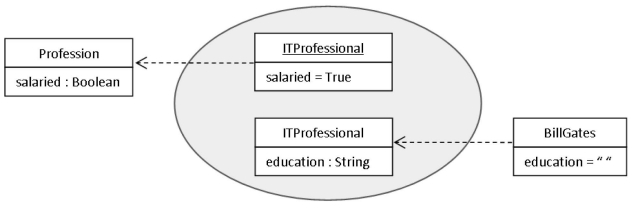

Fig. 4. Two viewpoints of profession.

and Gitzel and Schwind [22] have also proposed "nonlinear" metamodeling framework built around the distinction between these two forms of classification. However, a weakness of all these proposals is that they violate the fundamental principle of strict metamodeling in some way. This principle—a foundation of the UML infrastructure—holds that only instanceOf relationships should cross metalevel boundaries and that all instance of relationships should cross a level boundary [17]. Without this principle, the definition of model levels becomes somewhat arbitrary.

2.2 Class/Object Duality

Issue. When a user model only includes two ontological levels, each model element can play either the role of a type or the role of an instance but not both.

However, as soon as there are more than two ontological levels, the elements in the middle level (other than the top and bottom level) play two roles—they are both instances of the ontological level above and types for the ontological level below. This is illustrated in Fig. 4 which shows how the notion of *ITProfessional* can be seen from two viewpoints. The upper part of the ellipse represents the object view of *ITProfessional* when seen as an instance of *Profession*, while the lower part represents the class views of *ITProfessional* when seen as the type of the object *BillGates*.

Current status. The convention adopted in mainstream modeling languages and tools is to only show one facet of dual-faceted model elements (i.e., model elements with both an instance and a type facet) at a time. In other words, in a given diagram, either the type facet or the instance facet of a model element is rendered but not both. The UML has no concrete syntax for showing both facets in a single symbol, so UML-compliant tools are also forced to adopt this convention.

Our solution. Since both facets of a dual-faceted model element are equally valid and important, we use a unified modeling concept to emphasize the duality of model elements and support it with a concrete notation for representing both facets at the same time within a single symbol. We use the term "clabject" (a unification of the terms class and object), introduced by Atkinson [17] and adopted in numerous other approaches and standards [13], to refer to dual-faceted model elements.

The proposed concrete syntax is shown in Fig. 5. This shows how the clabject, *ITProfessional*, can be represented as a single symbol. The type facet is represented by the attribute types such as *education*, while the instance facet is represented by attribute instances[3] such as *salaried*. In this simple example, the difference between an attribute type and an attribute instance can be determined by whether a type or a value appears after the name, but a more general solution will be discussed in the next section.

3. Also referred to as slots in previous papers, based on the Smalltalk terminology for attribute instances (i.e., values).

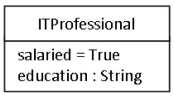

Fig. 5. Clabject.

Other solutions. The power-type pattern introduced by ODell [23] was one of the first attempts to systematically characterize and model the type and instance facets of model elements. It basically describes the two facets in distinct symbols, but tries to express the fact that they are somehow facets of the same abstraction. UML2.0 supports this by means of the "power-type" stereotype. Gonzalez-Perez and Henderson-Sellers have built on the power-type idea and elaborated it into a general approach for modeling dual-faceted elements [21]. While this has the advantage of requiring no new notational concepts, it has the disadvantage that the single concept is represented by two separated parts and, more seriously, these parts occupy different model levels according to the strict modeling doctrine. A similar notion to power types, known as materialization, was proposed by Pirotte et al. [24] for database modeling.

2.3 Deep Classification

Issue. The ability to explicitly represent domain concepts that occupy multiple ontological classification levels brings with it the opportunity to describe subtle relationships and constraints that span multiple levels. Consider the example in the previous figures. Specific professions (instances of *Profession*) that are salaried usually have average or default salaries that are representatives of the field as a whole, while specific instances of specific professions (i.e., individuals working in a specific profession) can obviously have their own salary. This means that not only are all instances of *Profession* required to have a *salary* attribute, all instances of these instances are as well. This is a domain property which has been referred to as deep characterization [25], and which should ideally be supported in a multilevel modeling language.

Current status. To our knowledge, no existing modeling tool directly supports deep classification in the way described in this section.

Solution. Deep instantiation, suggested by Atkinson and Kühne [10], introduces the notion of potency as a measure of the endurance of a model element or its properties over instantiation steps. Like multiplicity constraints, potencies either have a nonnegative Integer value or a special "unlimited" value represented by a "$*$". Except for unlimited potencies, instantiation always lowers the potency of a model element or property by one. Thus, a model element or property of potency 0 is like an object or attribute instance because it cannot be further instantiated, while a model element or property of potency 1 is like a class or an attribute type because it can be instantiated exactly once. However, potencies of 2 or greater give rise to new kinds of model elements which can be instantiated more than once. An instance of an unlimited potency model element or property can itself have unlimited potency or a potency with a specific numeric value.

In Fig. 6, potency is shown as a small superscript after the name of the affected element. Thus, *Profession* is a clabject of

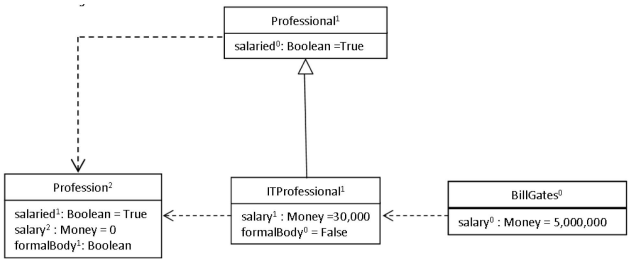

Fig. 6. Potency notation.

potency 2, and thus, can be instantiated twice (i.e., it can have instances, and these instances can have instances). *ITProfessional*, as an instance of *Profession*, has a potency of 1 and *BillGates*, as an instance of *ITProfessional*, has a potency of 0 and cannot be further instantiated. It is important to note that these potencies represent domain constraints, like multiplicity constraints on associations. They therefore help to make a model a more accurate representation of the domain it is modeling. It is possible to always use model elements with unlimited potencies that can be instantiated an unlimited number of times, in the same way that all associations in a UML model can be given the most general "zero or more" multiplicity (the default). However, such models usually do not provide the most accurate representation of a domain.

The assignment of potencies to properties blurs the distinction between attributes (property types) and slots (property instances). For example, how should a property of potency 2 be interpreted, as a type or an instance? The potency assigned to each property captures its semantics, so it is no longer necessary to distinguish between property types and property instances based on whether they have a type or value after the name, as described in the previous section. Instead, properties can be modeled using a single unified concept which we refer to as a "field." In general, a field has three parts: a name, a type, and a value. For fields of potency 0, the latter is simply the local value of the property for that particular instance, but for fields of potency 1 or greater it is normally viewed as a default value.

Other solutions. Building on their solution to the class/object duality problem, Gonzalez-Perez and Henderson-Sellers suggest the use of the power-type pattern to support deep classification [21]. However, this has the disadvantage that it requires a distinct model element to be defined for each facet, and these elements are conceptually at different ontological metalevels. Strictly speaking, their approach fails to satisfy the strictness principle because, in order to keep these two facets physically at the same level, they have to mimic the classification relationship between them using an association (similar to the type instance pattern of Johnson and Woolf [9], for example).

2.4 Uniform Connector Structure and Visualization

Issue. Although the concepts of clabjects and deep instantiation (or something similar) are necessary for creating a language that supports multilevel modeling in a uniform and systematic way, they are not sufficient. Another fundamental issue that needs to be addressed is the way in which connectors are supported. We use the term "connector" here to represent any information that is usually represented as an edge in graph-based models. In the UML, this includes associations, compositions, links, generalizations, and dependencies.

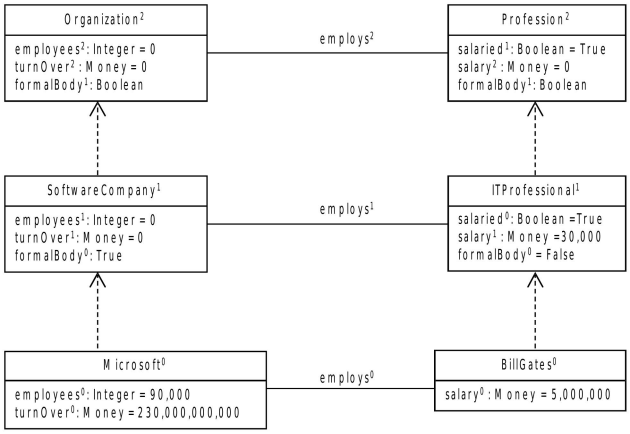

Fig. 7. Relationships as lines (edges).

The root of the problem is the type/instance duality of elements in the middle ontological model levels in the OCA. The clabject concept supports this duality for classes/objects (i.e., concepts typically represented as nodes in graphical representations of models) but does not address all the issues involved in supporting connectors in a level-agnostic way. Just like nodes, relationships in a model also generally have a type/instance duality. For example, consider the middle *employs* relationship in Fig. 7 which shows the relation of the *Profession*, *ITProfessional*, and *BillGates* clabjects from the previous figures to a similarly related set of clabjects representing potential employers.[4] The middle *employs* relationship is both an instance of the *employs* relationship at the ontological level above and a type for the *employs* relationship at the ontological level below. In UML terminology, it is both a link and an association.

The issue arises because modeling languages often use a different structure to represent relationship types than to represent relationship instances. Although associations are generally rendered as a single line (with appropriate adornments), their internal structure is much more complex. According to the UML metamodel [1], a single association such as *employs* is represented using a collection of model elements, as illustrated in Fig. 8, a model element representing the association itself and two model elements represent the association ends.

Links, on the other hand, are generally represented internally using a single model element [26]. According to the UML, therefore, relationships such as employs in the middle ontological level would have to be represented in two different ways—one using the association approach corresponding to the type facet and the other using the link approach corresponding to the instance facet. To make matters worse, these representations often have a nonintuitive relationship to the graphical rendering of the connector. For example, as Fig. 8 shows, in the UML, the fully elaborated structure of an association (in terms of objects and links) is an elaborate subgraph consisting of multiple nodes and edges.

4. We have arranged the three ontological level vertically rather than horizontally to simplify the presentation. The meaning remains the same.

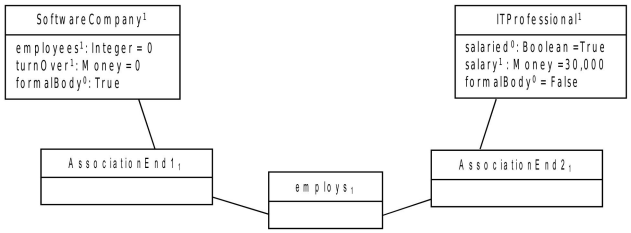

Fig. 8. Instance view of a UML association.

Current status. We are not aware of any tool that explicitly recognizes and supports dual-type and instance facets of connectors.

Our solution. In order to cleanly unify the type and instance facets of relationships, it is clearly necessary to have a single internal representation and a simple, coherent metaphor for rendering them that reflects this structure. To achieve this we believe that a multilevel modeling language should employ the following three principles for realizing connectors [26]:

Principle 1: Uni-element connector representation.

The information conveyed by a relationship should be captured by one clabject and only one clabject. Regardless of whether a relationship is just a type, just an instance or dual faceted, it should always be represented by a single clabject. This ensures that there is a single representation format for all relationships and provides the basis for a simple rendering approach.

Principle 2: Multiform connector rendering.

The first principle ensures that all connectors are represented internally in the same way using a single internal model element but it does not deal with their rendering. It therefore needs to be accompanied by a suitable principle for rendering connectors.

Since connectors are usually rendered using lines (edges) in modern, graphical modeling languages (indeed, this is, of course, the very meaning of connectors in a visual sense), we need a strategy to assign lines to the single model element that represents a connector according to principle 1. A simple way to do this would be to simply say that a connector is always rendered as a single line in a way that resembles the traditional UML rendering of association and links, as shown in Fig. 7. The problem is that it is also often desirable to be able to view a connector as a normal model element which can have attributes and connectors to other elements just like any other. In other words, to use a UML term, it is often desirable to view a connector as an association class.

To view a connector in this way, the natural approach is to render each connector as a normal model element using the usual rectangular notation for model elements, and in addition to draw two lines connecting them to their source and target elements. This approach is illustrated in Fig. 9. However, this not only diverges from normal rendering conventions, it also has the problem that the connectors are being visualized as nodes (normal model elements) in the diagram, and the questions arise as to how the two lines connecting them to their source and target should be interpreted. Are they connectors in their own right, and if so do they have their own model elements to represent them?

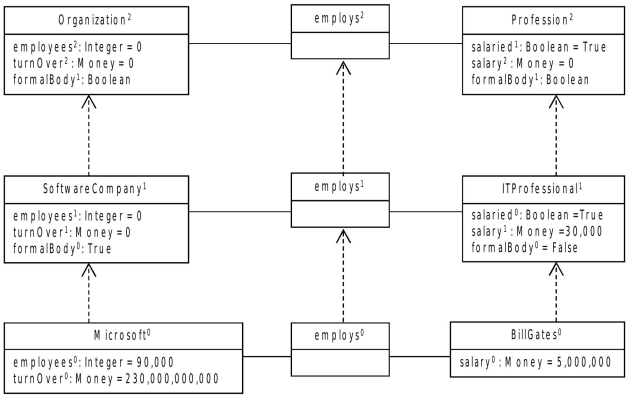

Fig. 9. Relationships as clabjects (nodes).

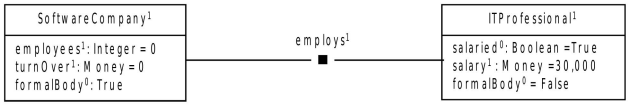

Fig. 10. Relationships as dots.

The infinite regression that this implies clearly indicates they cannot be regarded as connectors.

The solution we propose essentially aims to balance these two approaches. The principle is that all (binary[5]) connectors are always rendered in the form of a nodal symbol plus two lines (as in the second approach), but with the caveat that the node can take multiple forms, one of which can be "visually insignificant." By "visually insignificant" we mean that the model element can be rendered as little more than a dot which, although visible, gives the overall construction the visible appearance of an edge. This is illustrated in Fig. 10 which shows how the employs relationship between *SoftwareCompany* and *ITProfessional* can be rendered in a "visually insignificant" form. All of the connectors in Fig. 7 would be rendered using this technique in our approach. It follows that all edges in a diagram are purely visualization constructs that do not represent a connectors in their own right. Their visual form is controlled by the connector to which they belong (i.e., the *employs* relationship in this case).

This principle not only has the advantage that it represents all relationships in a uniform way, it allows them to be treated as first class citizens of a model in a simple and natural way. For example, in Fig. 9, *instanceOf* connections can also be shown between the relationships just as easily as they can be shown between node clabjects. Generalization connections between relationships can also be shown just as easily, which is not the case in the UML. From a UML perspective, this principle in effect requires all relationships to be treated as association classes, but without the awkward UML notation. It also provides a clean way for other rendering forms to be applied to connectors. For example, Fig. 11 shows a domain-specific rendering of Fig. 10 in which the *employs* relationship as well as *SoftwareCompany* and *ITProfessional* are assigned domain-specific symbols. This can be done in a conceptually clean way because a

5. The approach scales up naturally to higher order connector (e.g., ternary connectors, but to simplify the discussion, we focus on binary connectors here).

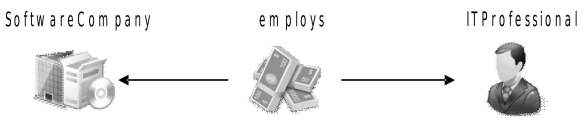

Fig. 11. Domain-specific rendering.

connector is always rendered as a node (even if sometimes, it is visually insignificant).

Principle 3: Exclusive and mandatory connector-based rendering of relationships.

The final principle deals with another source of confusion that often occurs with the rendering of models in infrastructures like the UML. Although the previous principle explains how connectors are rendered, it does not explain when they are rendered (i.e., should they be rendered in all circumstances) and whether there are any other forms of information in models that can be rendered as lines (i.e., edges).

There is certainly a lot of information within the normal (nonconnector) model elements that could potentially be rendered as edges. For example, every node has information about its classifier that could easily be rendered as an *instanceOf* arrow. But allowing any conceptual relationship stored in the model repository to be rendered as an edge at will (without actually being connectors in the sense described above) detracts from the clear rendering metaphor established by the previous two principles and makes the management of model visualizations much more of an ad hoc activity.

For this reason, we add a third principle to the previous two which requires that: 1) Only the rendering of connectors can include the use of lines and 2) the rendering of all connectors must include the use of lines. The first of these requirements is made practical by ensuring that the multilevel modeling language has a concrete syntax (i.e., a visualization approach) that allows internal relationships to be captured in the nodes of a rendered model without the use of edges. An example is the UML's notation for indicating the class of an object by placing it after a colon following the object's name. The practicability of the second requirement becomes clearer when one realizes that today, models are increasingly generated on demand from other models by automated transformations. Thus, in modern modeling tools a specific diagram can be generated on the fly from a central model for the specific purpose of creating a specific view of it. The transformation process can thus insert the necessary connectors as desired by the user for the specific diagram, but within the central model, the information used to generate the connector can remain within the original model elements. Thus, if a user wishes to create a diagram in which *instanceOf* relationships are actually rendered visually using arrows, he or she needs to direct the tool to generate the appropriate connectors which can then be rendered using the collapsed (dotted) form. In the context of such transformation-based modeling tools, there is no point in having any model elements which are not rendered. In other words, either a model is rendered in full or not at all.

Other solutions. We are not aware of any other proposals for explicitly supporting the dual-faceted nature of relationships and for allowing them to be visualized in a level-agnostic way.

2.5 Deep Constraint Language

Issue. A natural complement to a modeling language that accommodates multiple ontological classification using notions such as clabjects and deep instantiation is the ability to define constraints that are "aware" of ontological classification. For example, if it is a property of the domain that all the instances of the instances of a given concept satisfy a particular constraint, it should be possible to define it explicitly and concisely using the classification-related features of a constraint language (e.g., allInstances), rather than using a work around based on navigation across associations. Potencies provide a concise way of modeling constraints on the meaningful ontological instantiation depths that may exist in a domain, but they do not allow arbitrary constraints across multiple ontological levels to be expressed.

Current status. No existing DSM tool has a constraint language or checking mechanism that recognizes and explicitly supports ontological classification.

Solution. The solution to this requirement is an enhanced constraint language that is "aware" of the semantics of ontological classification and the concepts defined in the previous sections and allows constraints to span more than one ontological classification level. An example of a constraint in a "deep" constraint language of this kind is shown below:

$$\begin{aligned} &context\ Profession\ inv \\ &NonNegativeSalaries{:}self.allInstances().salary >= 0 \\ &\quad and\ self.allInstances().allInstances().salary >= 0. \end{aligned}$$

This is a constraint on the clabject *Profession* in Fig. 9, which states that the *salary* property of all instances of *Profession* must have a value greater than or equal to zero and the *salary* property of all instances of all instances of *Profession* must also have a value that is greater than or equal to zero. As mentioned previously, potencies actually represent a kind of constraint and could be expressed in a "deep" constraint language.[6]

Other solutions. We are not aware of any proposals for a model constraint language constraint language that recognizes or supports the distinction between ontological and linguistic classification or the expression of constraints that span multiple ontological levels.

2.6 Ontological Classification-Driven Visualization

Issue. Contemporary DSM tools generally assign domain-specific symbols to model elements based on their linguistic classifiers. As a result, all instances of the same linguistic metamodel element (i.e., the same language construct) are generally rendered with the same symbol. From a user's perspective, however, it is often a concept's ontological classifier that best determines how it should be rendered, not its linguistic classifier. We believe that DSM tools should therefore allow users to assign their own preferred symbols to model elements in a more flexible way based on their ontological as well as linguistic classifiers. More specifically, the default symbol derived from an element's linguistic type should be overrideable by an element's ontological classifiers. For example, a user may wish to specify symbols

6. In the same way that multiplicities in a regular UML class diagram can also be defined in OCL.

for the model elements at the bottom of Fig. 7, such as those in Fig. 11. If no specific symbols are assigned, the symbols for the model element's linguistic types would be used.

Current status. Some tools allow the ontological types of model elements to influence their rendering in certain ways. For example, GME [8] allows different icons to be defined for types, subtypes, and instances. It also supports the notion of decorators that allow the appearance of model elements to be driven by attribute values. However, the former is only applicable across one pair of ontological levels (since GME does not explicitly support more than two) and the latter is a somewhat ad hoc mechanism that is independent of the typing system.

Our solution. The explicit representation of multiple classification levels within a model provides a natural basis for working out which symbol to use to render a model element. We use a "deep" symbol assignment (or rendering) mechanism that resembles the algorithm used to find methods in a dynamic object-oriented language like Smalltalk. First, the symbol assigned to the individual element is used, if there is one. If not, the symbol assigned to the model element's ontological type is used, if there is one. If not, a symbol assigned to a supertype of the model element's ontological type is used (closest first), if there is one. If not, the symbol assigned to the ontological type of the ontological type of the model element's type is used, if there is one, and so on, until the whole ontological derivation tree of the model element has been traced. If no special domain-specific symbol is found, the linguistic classifiers are used to find a suitable symbol. Tools like GME [8] use a similar algorithm, but not across multiple ontological levels.

Allowing ontological and linguistic classification relationships to drive the rendering of models means that the domain-specific and general renderings of a model can be supported at the same time (i.e., users can choose to see a model in a domain-specific way or in a general way). For example, the domain-specific representation of *SoftwareCompany*, *employs*, and *ITProfessional* in Fig. 11 could be defined through ontological classification, while the general-purpose rendering shown in Fig. 7 could be defined through linguistic classification. This addresses one of the big problems of domain-specific languages which is their tendency to lower the communication value of diagrams—the so-called "Tower of Babel" problem. Supporting both domain-specific and a general-purpose renderings of models retains their communication value (through the general view) as well as their usability by domain experts (through the domain-specific view).

Other solution. Apart from GME, we are not aware of any tools or proposals for rendering mechanisms that recognize ontological as well as linguistic classification. Moreover, even including GME, we are not aware of any tools in which the rendering algorithm searches in a recursive and uniform way across multiple ontological classification levels.

3 Infrastructure

In the previous section, we outlined the main principles and techniques that we believe a language engineering environment should support in order to leverage ontological and linguistic classification across multiple levels. In this section, we present the detailed architecture of our prototype modeling infrastructure and explain how solutions to the aforementioned issues can be integrated into a unified approach. The infrastructure consists of four main predefined parts:

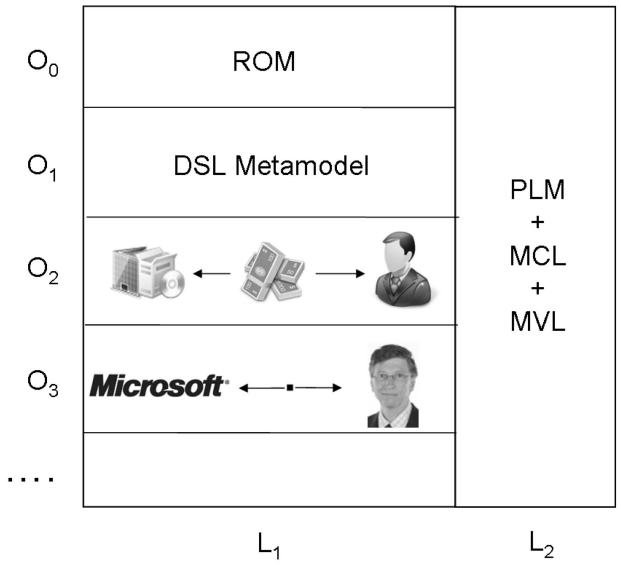

Fig. 12. Multilevel modeling architecture.

1. Pan-Level Metamodel (PLM);
2. Root Ontological Model (ROM);
3. Multilevel Constraint Language (MCL);
4. Multilevel Visualization Language (MVL).

The role of each of these within the OCA and their relationship to typical user-defined languages and models is depicted in Fig. 12. Since they provide the foundation for multilevel modeling, along with deep constraints and rendering, the PLM, MCL, and MVL represent the linguistic foundation for the infrastructure—that is, they represent L_2 in the OCA as depicted in Fig. 12. The ROM is the predefined foundation for the ontological modeling levels, and thus represents the start (or the root) of the ontological-level hierarchy within the L_1 linguistic level. We refer to this as O_0.[7]

Pan-level metamodel. The PLM is the foundation of the infrastructure that defines all essential concepts for multilevel, object-oriented modeling as defined in Section 2. The core of the PLM is depicted in Fig. 13. By convention, all the names of PLM elements begin with P. This figure shows the pure data model of the PLM, so no UML-like associations between types are drawn. This has the advantage that all the "baggage" associated with the semantics and rendering of associations from the UML is avoided, and the rendering of elements at L_1 can be built up from scratch.

In the PLM, there are two kinds of elements with dual class/instance facets, both of which are subtypes of *PClabject*. *PNode* is the element of the PLM which classifies "nonconnector" model elements with a class/object facet so that user-defined types can span multiple ontological levels. The other type that can span multiple levels is the subtype

7. Note that our numbering scheme for ontological levels is opposite to the "standard" one popularized by the UML infrastructure. The reason is to avoid the negative-level numbers that would occur with the UML approach when there are more than three ontological classification levels.

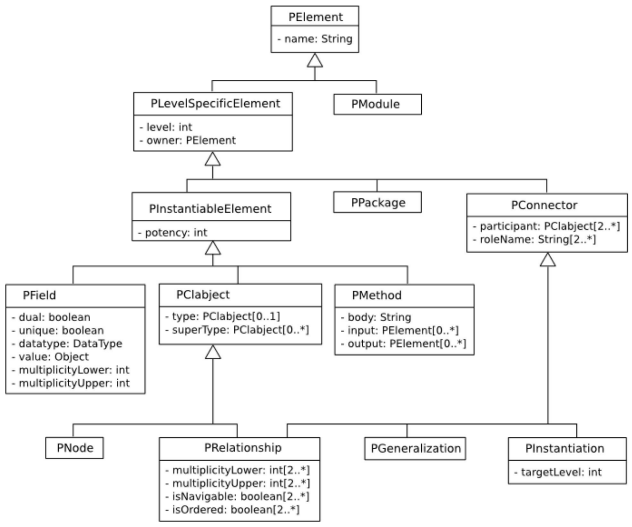

Fig. 13. PLM.

of *PConnector*, namely, *PRelationship*. This type enables the user to model relations between *PNodes* that can also span multiple ontological levels.

All subtypes of *PClabject* can have *PFields* which allow named and typed values to be associated with *PClabjects*, possibly spanning multiple ontological levels. The only other type that can span more than one level is *PMethod*. In many cases, it makes sense to define a method on an upper level and use it some levels below. In our example, the user could define a method to calculate the annual salary at level 1 in the *PNode, Profession*, and the calculation is performed at level 3.

All of the types which can be instantiated are subtypes of *PInstantiableElement*, which has the potency property. A *PInstantiableElement* can only be instantiated if its potency is "$*$" or is an Integer greater than zero, and the potency of the new element is either "$*$" or one lower than the potency of its type. Since the PLM is intended to provide the foundation for modeling with multiple ontological levels, the basic concepts of object-oriented development need to be supported. The PLM therefore explicitly supports the generalization and instantiation relationships. The user can model a *PGeneralization* or a *PInstantiation* relationship between two *PClabjects* and if the relationship is consistent with other information already in the model, the corresponding properties *superType* and *type* (see *PClabject*) can be set. Containment, as a fundamental relationship between objects, is also supported via the *owner* attribute.

Root ontological model. Because the OCA separates linguistic modeling concerns from ontological modeling concerns and provides built-in linguistic support for the latter, there is no constraint on the number of ontological models that can be supported. The user can define as many ontological levels as needed to best capture the domain of interest. However, as with all liner hierarchies of model levels, the question arises as to how to terminate (or start) the hierarchy? In the OMG model hierarchy, this is done by simply stating that the top most model (i.e., the MOF [27]) is regarded as being an instance of itself. However, few metametamodels actually present details of how this is done—that is, which elements of the top-level metamodel are instances of which other elements of the top-level metamodel.

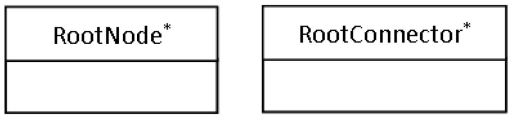

Fig. 14. Root ontological model.

Since our infrastructure allows an unlimited number of ontological levels, unlike the OMG's, the ROM does not have to be very complicated. Basically, the model at the root of the ontological-level hierarchy O_0 defines the minimum set of model elements needed to represent the basic types for all model elements at level O_1 and beyond. The form of the ROM is illustrated below in Fig. 14.

As with the elements in the MOF, the types of *RootNode* and *RootConnector* are not defined. However, since the ROM is predefined and fixed, this is not an issue that users of the infrastructure who define models at O_1 and above need worry about. When defining models at O_1, the predefined model elements in the ROM can be used as the types of the model elements, without concern for how they are created or what their own types are. Thus, the ROM provides a simple and stable foundation upon which users (e.g., DSL modelers) can base their own models and languages.

The "$*$" symbol for the potency of *RootNode* and *RootConnector* indicates that they have unlimited potency. In other words, there is no limit to the depth to which they can be instantiated. Unless explicitly stated otherwise, the potency of an instance of a model element with potency "$*$" is also unlimited. However, it can also be lowered to any nonnegative Integer as required by the properties of the domain being modeled.

Multilevel constraint language. The MCL is essentially an enhancement of the OCL that has an "understanding" of how to operate across multiple ontological levels. Standard OCL can be, and is, used to define static semantic constraints at the L_2 level, but any standard OCL constraint can only operate across linguistic levels—that is, between the definition of the language at the L_2 level and the use of the language at the L_1 level. Because the PLM uses the same basic classifier for all model elements, regardless of their ontological levels, OCL constraints written at the L_2 can apply to all L_1 clabjects. For example, the following constraint on the class *PinstantiableElement* in the PLM ensures that all model elements at L_1 have a nonnegative potency value, regardless of their position in the ontological hierarchy:

$$context\ PInstantiableElement$$
$$inv\ noNegativePotencies\colon potency >= 0.$$

The most fundamental feature that distinguishes the MCL from regular OCL is the ability of the *allInstances* operator to return all ontological instances of a clabject rather than all linguistic instances, dependent on the context in which it is used. Thus, the following constraint on the PLM class *PInstantiableElement* ensures that no clabject of potency zero has any ontological instances:

$$context\ PInstantiableElement$$
$$inv\ noOntologicalInstances\colon self.potency = 0$$
$$implies\ self.allInstances() ->size() = 0.$$

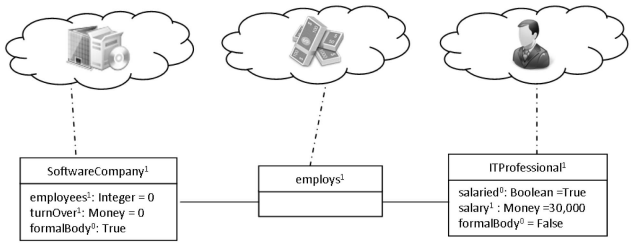

Fig. 15. Attaching symbols to elements.

Another important distinction is that MCL constraints can also be applied to L_1-level model elements to control their ontological instances. This would not make sense in a standard modeling environment because L_1 elements have no instances as far as the linguistic typing hierarchy on which standard OCL operates are concerned. For example, the following constraint on the *Profession* clabject in Fig. 9 states that its ontological instances must have a *salary* property whose value is greater than or equal to 30,000 and that all ontological instances of its ontological instances must have a salary property greater than 30,000:

$$context\ Profession$$
$$inv\ maximumWage\text{: } allInstance.salary >= 30{,}000\ and$$
$$allInstances.allInstance.salary >= 30{,}000.$$

This constraint is therefore "aware" of the deep instantiation mechanism described in Section 2.3. It not only constrains the (ontological) instance of *Profession*, but also the (ontological) instances of its instances.

Multilevel visualization language. The MVL and supporting algorithms are the part of the infrastructure that allows new domain-specific symbols to be assigned to model elements based on their ontological as well as linguistic classifiers. The actual notation used to define the concrete rendering of a model element (the concrete syntax) is similar to that used by GMF [5]. However, for space reasons, we provide no further details here. Whatever syntax is actually used to make the assignment, the goal is to associate a compound symbol with each model element so that it can be rendered. In Fig. 15, we symbolize this effect by attaching a cloud to a model element containing the symbol that has been associated with it. We refer to the symbols as "compound" because they also accommodate the displaying of the name of the model element and properties captured by the fields, etc.

One of the key features of the MVL is the algorithm used to locate the particular symbol to be used to render a model element. Because all PLM elements have predefined symbols associated with them, the algorithm is bound to find a suitable symbol for every mode element. Falling back to the PLM symbols is the semantically worst case because the symbol chosen is then the least "domain-specific." Informally, the algorithm operates as follows:

1. If the model element has a symbol directly defined for it, use that symbol.
2. If not, if the model element has a supertype and that supertype has a symbol defined for it, use that.
3. If not, continue searching up the supertype hierarchy of the model element and use the first symbol found.

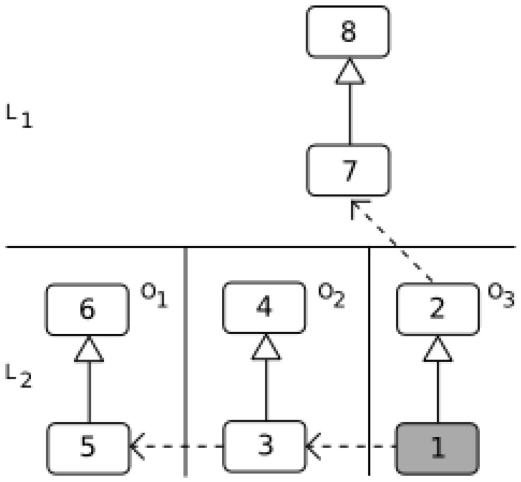

Fig. 16. Search order of rendering algorithm.

4. If no symbol is found, if the ontological type of the model element has a symbol directly defined for it, use that symbol.
5. If not, search up the supertype hierarchy of the model element's ontological type, and use the first symbol found.
6. If no symbol is found and the ontological type of the ontological type of the model element has a symbol directly defined, use that one.
7. If not, repeat 5 for the ontological type of the ontological type of the model element.
8. If no symbol is found, repeat 5 and 6 for the ontological type of the ontological type of the ontological type of the model element, and so on, up the model element's classification hierarchy.
9. If no symbol has been found and all levels have been searched, search through the linguistic classifiers using the same algorithm.

Fig. 16 provides a visualization of how this algorithm works. The numbers inside the rectangles give the order in which the algorithm visits model elements when trying to find a symbol to render the shaded model element. The elements in the top L_2 level are in the PLM. The algorithm starts with the element itself, 1, and then, proceeds to search up the supertype and type hierarchies at the L_1 level. The L_2 level (PLM) is only considered if no symbol is found at the L_1 level. Since the elements in the PLM have rendering symbols associated with them, the algorithm is ultimately bound to find a suitable symbol for any L_2 element.

4 Examples

In this section, we show two small examples that illustrate the advantages of the infrastructure described in the paper. The first comes from the domain of "head hunting," where a system needs to know who is working for what company for what salary and what additional benefits might attract them such as their leisure pursuits and hobbies. The second is a small language for representing state machines much like that in the UML.

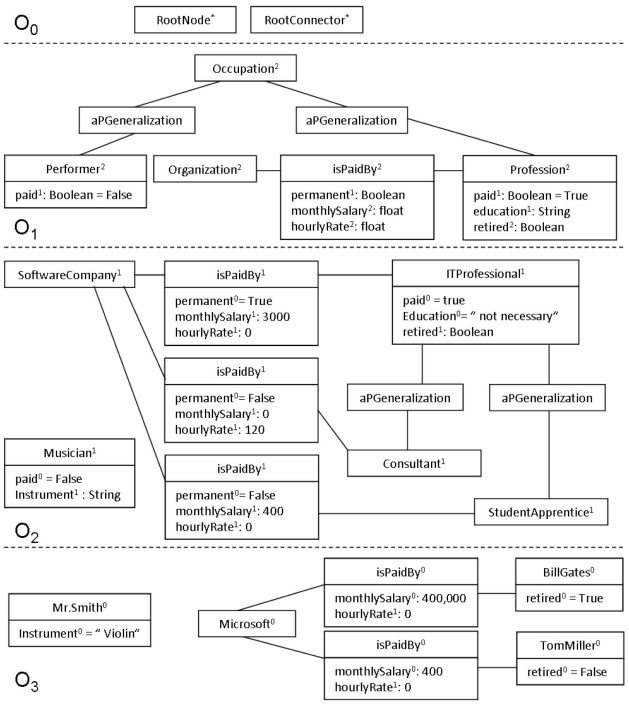

Fig. 17. A standard PLM rendering of the headhunter example.

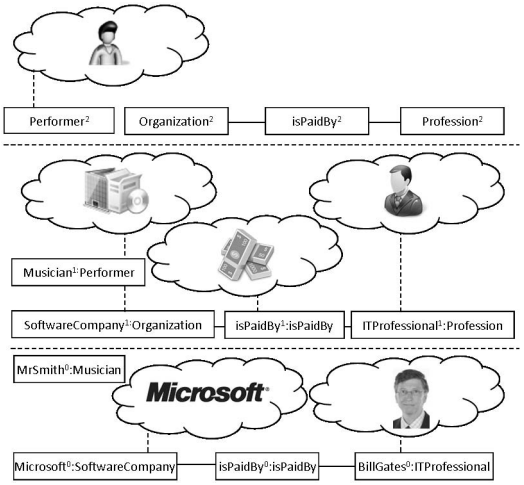

Fig. 18. Assignment of domain-specific symbols in headhunter example.

4.1 Headhunter

This example models the kind of information that a headhunter would be interested in maintaining. One standard PLM rendering is shown in Fig. 17. In this rendering, all connectors are shown in exploded form. Note that the model contains two instances of the *PGeneralization* PLM element that explicitly shows generalization relationships between *ITProfessional* and two of its subclasses, *Consultant* and *StudentApprentice*.

Fig. 18 symbolically shows how symbols can be assigned to the elements of the headhunter model. Only *Performer* has an associated symbol on level O_1. Some symbols are added at level O_2 and because of their special status, new symbols are assigned to two objects at level O_3.

Fig. 19 shows how the information at ontological level O_3 would be rendered based on this assignment of symbols. Despite the fact that two of the objects at this level have no directly assigned symbol, they are nevertheless rendered in a domain-specific way due to the deep rendering algorithm described in Section 3.4. The symbol for the *isPaidBy* relationship is derived from its direct ontological type, while the symbol for *MrSmith* is derived from its type's type.

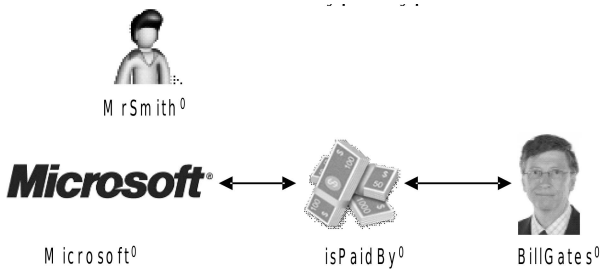

Fig. 19. Domain-specific rendering of O3 in the headhunter example.

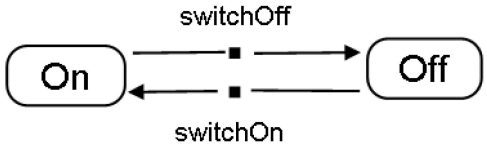

Fig. 20. Domain-specific rendering of the state machine example.

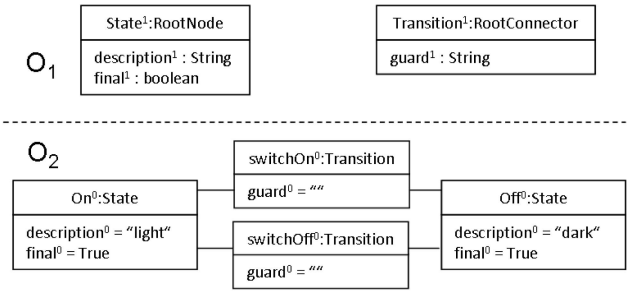

Fig. 21. PLM rendering of the state machine example.

4.2 State Machine Modeling

So far, we have only shown examples of structural models (i.e., models that shows classes, properties, and the relationships between them). As an example of a nonstructural model, in this section, we show how a small DSL for a state machine model could be defined. The example is a simple state machine for a desktop light switch. It has two states, "on" and "off," and two transitions between them. Fig. 20 shows the model in the DSL representation.

A standard PLM rendering of the state machine is shown in Fig. 21. There are two *PClabjects* at the ontological metalevel and four instances. The *Transition PClabject* is a *PRelationship* and the *State PClabject* is a *PNode*.

To define the domains-specific rendering of this model, appropriate symbols have to be assigned to the relevant model elements. This information is assigned to the clabjects *State* and *Transition* as shown in Fig. 22.

5 Related Work

Although the basic idea of the OCA was put forward by Atkinson and Kühne over seven years ago [10], to our knowledge, no group has carried this forward to a full working prototype of a modeling and language engineering environment. We believe that the work reported in this paper is the first attempt to do so. Nevertheless, the

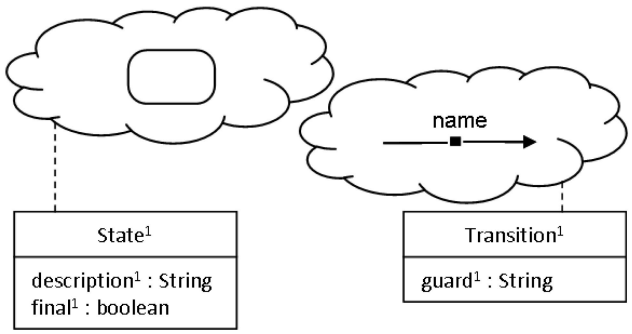

Fig. 22. Symbol association in the state machine example.

approach does overlap with other approaches reported in the literature.

As described in the original proposal for the OCA by Atkinson and Kühne [11], there are a number of environments that do allow a multilevel approach to modeling in which classification is a first-class element of the modeling language and multiple classification levels are supported. Some well-known examples include ConceptBase [28], Telos [29], and Ptolemy [30]. However, these approaches have two differences to that described in this paper. First, they do not enforce a "strict" approach to metamodeling in which the modeling space is organized into well-defined model levels based on the ontological *instanceOf* relationship. In a sense, they advocate a freestyle approach in which *instanceOf* relationships can be mixed arbitrarily with other relationships to yield a kind of "soup" of model elements. Second, they do not attempt to support a generalized UML style notation alongside user defined, domain-specific notations so that models can be viewed using two notations.

Over the last few years, another high profile language has been added to this group—OWL [31]. Although the simpler version of OWL (OWL Lite) is very much based on a standard two-level modeling approach (classes and instances), OWLFull at least conceptually allows an element to behave as a class and an object. However, OWLFull is rarely if ever used in the ontology community and the standard syntax used to represent OWLFull models is XML. The only graphical notation is the highly simplistic RDF rendering notation supported by tools like Protégé [32] based on ellipses and lines. It is interesting to note that many ontology developers use the UML to visualize ontologies despite the fact that there are some fundamental mismatches between the two languages [33]. Also, like in Concept Base [28], OWLFull has no notion of a "strict" approach to organizing model elements.

As mentioned previously, Kühne and Schreiber have applied the notion of deep classification at the implementation level in an environment they call DeepJava [15]. This allows users to define Java classes which are instances of, as well as subclasses of, other Java classes. Such classes can then inherit their type facet from their superclasses and their instance facet from their type. Although this is an interesting approach, it is not intended to support visual modeling or the definition of domains-specific languages.

Perhaps the work that comes closest to our work, at least in intent, is that of Gonzalez-Perez and Henderson-Sellers [13]. They have built a tool which is based on the core notion of the OCA in that it provides a modeling language that allows multiple *instanceOf* relationships to exist within a model. It also recognizes the importance of organizing model elements into distinct levels and supports the notion of clabjects with a distinction between the type and object facets of model elements. However, the notion of model levels supported in the tool is not based upon ontological *instanceOf* relationships, as the principle of strictness would require, and clabjects are not represented using a single unified notion but using the power-type concept where the type facets are represented separately. The tool's unusual approach for defining model levels results from its attempt to allow the two facets of clabjects to occupy the same level, whereas the tenets of strict metamodeling would require them to occupy different levels. We believe that this makes the approach somewhat counterintuitive.

6 Discussion

We believe that there are numerous advantages to the approach outlined in this paper. The most important three are the following:

1. It combines the advantages of a mature general-purpose modeling language such as the UML with the advantages of domain-specific modeling languages supporting user-defined, domain-specific visualizations and structuring rules.
2. It allows connectors to be truly first-class elements of a model for the first time, by cleanly bringing them into the multilevel modeling approach and unifying their type and instance facets in a clean and coherent way.
3. It provides an ideal foundation for integrating and enhancing technologies based on representing the relationship between instances and their types. It does this by providing a uniform representation of multiple ontological classification levels and allowing instances to be modeled along with their types with equal status.

The first advantage allows a model to be viewed in two distinct languages, whichever the modeler finds the best for a particular purpose. This is achieved by effectively making the definition of visualization choices (concrete syntax) and structuring choices (static semantic rules) part of what traditional DSM tools would regard as the (user-defined) model rather than part of the language. These may be placed into distinct "ontological" levels, but they are nevertheless part of the "model" from the linguistic metamodeling point of view. In short, they effectively make "language engineering" part of the model rather than part of the definition of the underlying language's abstract syntax.

It is up to each software engineering company to decide whether the normal modeler has full control over all ontological levels, or whether the upper ontological levels are predefined by the chief architects and are simply instantiated by the normal users. By predefining and standardizing the upper ontological levels, companies can fix their own company-specific languages which their engineers can then use at lower ontological levels. Indeed, as Atkinson and Kühne have pointed out in several papers [34], [35], this infrastructure would greatly simplify the definition of the UML itself. As the second-state machine example readily demonstrates, the UML modeling concepts and diagram types (i.e., essentially, the UML superstructure)

could easily be defined at the O_1 level of the OCA as straightforward ontological metamodels with the appropriate usage constraints and symbols. This would also make the UML easily extensible and customizable at the O_1 level by standard subtyping (specialization in UML terminology) and allow the full intent of the UML profiling mechanism (user-friendly UML customization) to be supported without all the complex and counterintuitive baggage of stereotypes. It would also do away with all the strange contortions that are currently performed at the M_2 and M_3 levels of the UML infrastructure to align the MOF and UML standards, where the UML metamodel is both a subtype of, and an instance of, the same set of core model elements (the InfrastructureLibrary) [2].

The second advantage effectively allows all relationships to be treated as association classes, but without the clumsy notation of the UML. In particular, relationships can

- have attributes and relationships of their own;
- be the source and target of subtyping and *instanceOf* relations;
- have deep characterization properties like nodal clabjects.

In short, the approach makes relationships fully blown clabjects in their own right but with special rendering conventions (in the PLM, *PRelationship* is a subclass of *PConnector*). The "trick" of always requiring connectors to be rendered as nodes, with the caveat that this node can be a visually insignificant dot, actually has profound consequences. When accompanied by the requirement that all visible relationships be modeled by a single connector, it ensures that there is only one unique interpretation of a model as a graph. This is not the case in current general-purpose environments like the UML and EMF, where models always have two graphical interpretations, one in terms of their "external" rendering using the UML concrete syntax for types, and the other in terms of their "internal" representation as UML metamodel instances. Typically, the former is the graph seen by users, while the latter is the graph recognized by tools. These two usually differ because some concepts (like associations) have a complex internal representation but a simple graphical rendering. The existence of more than one graphical interpretation of a model is obviously a recipe for confusion and error when, for example, defining metrics or measuring the quality of models. The approach outlined in this paper avoids this problem because it ensures that there is only one graphical interpretation of a model.

Making relationship types first-class citizens of the language whose instances can themselves be relationship types overcomes a common misuse of the UML when used to define metamodels. When defining a metamodel (or a domains-specific language), it is natural to want to visualize the type of a relationship as a relationship. This is, in fact, what many users do when they define metamodels in UML —they include associations in the metamodel which they intend to be the type for associations in models instantiated from the metamodel. However, this is strictly illegal in UML. An instance of an association in UML is a link, which not only has the semantics of an instance rather than a type, but it has a completely different representation structure from associations. This problem is addressed in our infrastructure since associations, links, and indeed, all relationships have the same internal representation—a single clabject.

The third advantage provides a way for modeling tool to provide some of the advanced reasoning services of ontology tools, and ontology tools to provide some of the visualization flexibility and power of modeling tools. More specifically, we believe that the infrastructure proposed in this paper provides an ideal foundation for integrating the "open world," knowledge discovery metaphor of ontologies and ontology tools (e.g., Protégé [32]), with the "closed world," knowledge application metaphor of models and modeling tools such as those centered on the UML. Since the infrastructure can represent and store instances (i.e., resources in ontology terminology) alongside their ontological types (i.e., classes in ontology terminology) in a uniform way, and can do this across multiple ontological classification levels as supported in OWL Full, the chronological order in which knowledge is added to the repository is immaterial. In other words, it is irrelevant whether instances are created from and checked against types or whether types are "discovered" from instances. As long as the underlying frame assumption in use at a particular point in time is visible to the user, and can be changed by the user as needed, both styles of modeling can be applied to the same "knowledge base." Moreover, since the approach allows sophisticated general-purpose modeling notations (e.g., UML-like) and domains-specific modeling notations to be applied to knowledge bases across arbitrarily many ontological classification levels, it can solve one of the perennial problems of the ontology modeling/semantic Web community—the lack of a sound and user-friendly concrete notation.

7 Conclusion

Since the infrastructure described in this paper is based on a different architecture to existing language engineering tools, and incorporates approaches to modeling and rendering that have not been implemented before, the creation of a prototype is a challenging endeavor. Therefore, our research strategy is based on the design sciences paradigm outlined by Hevner et al. [37]. This revolves around the demonstration of prototype solutions backed up by sound argumentation and empirical investigations. Once we have attained a stable version of the prototype tool, we therefore plan to perform experiments on user acceptance levels and user effectiveness using the approach.

The current version of the tool is being implemented in Java, using EMF/Ecore technology as the basis for the repository and GMF as the basis for the rendering engine. Thus, the starting point for our current implementation was the realization of the PLM as an Ecore model. Other implementation approaches are, of course, possible as well. In fact, any platform that supports the PLM would be satisfactory. In order to reinforce the overlap with ontology technology, we are currently developing an implementation based on RDF as the underlying PLM storage medium. Another approach is to map the PLM types directly to database schemas.

Given the claimed advantages of the approach, it is tempting to ask why this approach has not been tried before. We believe that one reason is the "if it ain't broke —don't fix mentality" in the DSM community, which takes

the view that since current language engineering and modeling tools clearly "work" and deliver clear benefits, why is a fundamentally new infrastructure needed. It is certainly true that DSM tools have made significant strides in recent years and now deliver impressive features to end users. However, as we have explained in this paper, there are a lot more services that a language engineering and modeling tool could provide. We hope this paper and the prototype infrastructure that it describes will help deliver these services to users.

REFERENCES

[1] OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2, OMG Document Number: Formal/2007-11-03, 2007.
[2] OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2, OMG Document Number: Formal/2007-11-01, 2007.
[3] *Proc. First Int'l Conf. Software Language Eng.,* Sept. 2008.
[4] *Proc. Eighth OOPSLA Workshop Domain-Specific Modeling,* Oct. 2008.
[5] Graphical Modeling Framework, http://www.eclipse.org/modeling/gmf/, 2009.
[6] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* John Wiley and Sons, 2004.
[7] J. Tolvanen, "MetaEdit+: Domain-Specific Modeling for Full Code Generation Demonstrated," *Proc. 19th Ann. ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications,* Oct. 2004.
[8] A. Lédeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *Computer,* vol. 34, no. 11, pp. 44-51, Nov. 2001.
[9] R. Johnson and B. Woolf, "Type Object Pattern," *Pattern Languages of Program Design 3.* Addison-Wesley, 1997.
[10] C. Atkinson and T. Kühne, "The Essence of Multilevel Metamodeling," *Proc. Fourth Int'l Conf. Unified Modeling Language, Modeling Languages, Concepts, and Tools,* Oct. 2001.
[11] C. Atkinson and T. Kühne, "Rearchitecting the UML Infrastructure," *ACM Trans. Modeling and Computer Simulation,* vol. 12, no. 4, pp. 290-321, 2002.
[12] M. Gogolla, J.-M. Favre, and F. Büttner, "On Squeezing M0, M1, M2, and M3 into a Single Object Diagram," *Proc. Model Driven Eng. Languages and Systems Workshop Tool Support for OCL and Related Formalisms,* 2005.
[13] C. Gonzales-Perez and B. Henderson-Sellers, *Metamodelling for Software Engineering.* John Wiley and Sons, 2008.
[14] D. Gašević, N. Kaviani, and M. Hatala, "On Metamodeling in Megamodels," *Proc. ACM/IEEE 10th Int'l Conf. Model Driven Eng. Languages and Systems,* 2007.
[15] T. Kühne and D. Schreiber, "Can Programming Be Liberated from the Two-Level Style?—Multi-Level Programming with DeepJava," *Proc. ACM SIGPLAN Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications,* Oct. 2007.
[16] UML 2.0 OCL Specification, OMG Adopted Specification ptc/03-10-14, 2003.
[17] C. Atkinson, "Metamodeling for Distributed Object Environments," *Proc. First Int'l Workshop Enterprise Distributed Object Systems,* 1997.
[18] R. Geisler, M. Klar, and C. Pons, "Dimensions and Dichotomy in Metamodeling," *Proc. Third BCS-FACS Northern Formal Methods Workshop,* Sept. 1998.
[19] J. Bezevan and O. Gerbe, "Towards a Precise Definition of the OMG/MDA Framework," *Proc. Ann. Int'l Conf. Automated Software Eng.,* Nov. 2001.
[20] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe, "The Architecture of a UML Virtual Machine," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications,* 2001.
[21] C. Gonzalez-Perez and B. Henderson-Sellers, "A Powertype-Based Metamodeling Framework," *Software and System Modeling,* vol. 5, no. 1, pp. 72-90, 2006.
[22] R. Gitzel and M. Schwind, "Using Non-Linear Metamodel Hierarchies for the Rapid Development of Domain-Specific MDD Tools," *Proc. 10th IASTED Int'l Conf. Software Eng. Applications,* 2006.
[23] J. Odell, "Power Types," *J. Object-Oriented Programming,* vol. 7, no. 2, pp. 8-12, 1994.
[24] A. Pirotte, E. Zimányi, D. Massart, and T. Yakusheva, "Materialization: A Powerful and Ubiquitous Abstraction Pattern," *Proc. 20th Int'l Conf. Very Large Data Bases,* Sept. 1994.
[25] T. Kühne and F. Steimann, "Tiefe Charakterisierung," *Proc. Modellierung '04,* Mar. 2004.
[26] M. Gutheil, B. Kennel, and C. Atkinson, "A Systematic Approach to Connectors in a Multi-Level Modeling Environment," *Proc. 11th Int'l Conf. Model Driven Eng. Languages and Systems,* 2008.
[27] Meta Object Facility (MOF) Core Specification Version 2.0, OMG Document Number: Formal/06-01-01, 2006.
[28] M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, M. Staudt, and S. Eherer, "ConceptBase—A Deductive Object Base for Meta Data Management," *J. Intelligent Information Systems,* vol. 4, no. 2, pp. 167-192, 1995.
[29] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis, "M.: Telos—A Language for Representing Knowledge about Information Systems," *ACM Trans. Information Systems,* vol. 8, no. 4, pp. 325-362, 1990.
[30] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming Heterogeneity—The Ptolemy Approach," *Proc. IEEE,* vol. 91, no. 2, pp. 127-144, Jan. 2003.
[31] OWL, http://www.w3.org/2004/OWL, 2004.
[32] Protégé Tool, http://protege.stanford.edu, 2008.
[33] K. Kiko, "Towards a Unified Knowledge Representation Framework," master's thesis, Univ. of Mannheim, 2005.
[34] C. Atkinson and T. Kühne, "Concepts for Comparing Modeling Tool Architectures," *Proc. Eighth Int'l Conf. Model Driven Eng. Languages and Systems,* 2005.
[35] C. Atkinson and T. Kühne, "Reducing Accidental Complexity in Domain Models," *Software and Systems Modeling,* vol. 7, no. 3, pp. 345-359, 2008.
[36] Eclipse Modeling Framework (EMF), http://www.eclipse.org/modeling/emf/, 2009.
[37] A.R. Hevner, S.T. March, J. Park, and S. Ram, "Design Science in Information Systems Research," *MIS Quarterly,* vol. 28, no. 1, pp. 75-105, 2004.

Colin Atkinson received the PhD degree in computer science from Imperial College, London. He holds the Chair of Software Engineering at the University of Mannheim, where his research interests focus on modeling and the use of component technologies to develop dependable software systems. He is a member of the IEEE.

Matthias Gutheil is a software architect at itemis AG. Before, he was a research and teaching assistant at the Chair of Software Engineering at the University of Mannheim. His research interests are model-driven development in particular multilevel modeling.

Bastian Kennel received the diploma in computer science from the University of Mannheim. He is a research assistant at the Chair of Software Engineering at the University of Mannheim. His research interests are multilevel constraint languages and metamodeling.