The Essence of Multilevel Metamodeling

Colin Aktinson and Thomas Kühne

AG Component Engineering
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
{atkinson,kuehne}@informatik.uni-kl.de

Abstract. As the UML attempts to make the transition from a single, albeit extensible, language to a framework for a family of languages, the nature and form of the underlying meta-modeling architecture will assume growing importance. It is generally recognized that without a simple, clean and intuitive theory of how metamodel levels are created and related to one another, the UML 2.0 vision of a coherent family of languages with a common core set of concepts will remain elusive. However, no entirely satisfactory metamodeling approach has yet been found. Current (meta-)modeling theories used or proposed for the UML all have at least one fundamental problem that makes them unsuitable in their present form. In this paper we bring these problems into focus, and present some fundamental principles for overcoming them. We believe that these principles need to be embodied within the metamodeling framework ultimately adopted for the UML 2.0 standard.

1 Introduction

The UML is an object-oriented language for describing artifacts in some domain of interest to the user. Like all languages, it needs to define the meaning of its descriptive concepts, the rules for putting them together, and the syntax to be used to represent them. In this respect, defining a graphical language to be used for modeling is conceptually no different to defining a textual language to be used for programming. Concepts like concrete syntax, abstract syntax and semantics are therefore also applicable for the definition of the UML.

For pragmatic reasons, early versions of the UML relied on natural language as the primary vehicle for describing the notation. However, in view of the well known limitations of natural language for this purpose, more recent versions of the UML have attempted to improve on this by using an abstract syntax accompanied by formal constraints. However, designing the abstract syntax for the UML has proven to be problematic in several ways [1].

Since the UML is a general purpose modeling language it is in principle suitable for describing its own properties. The motivation for using the UML to help describe its own semantics is the same as for any application of the UML—to provide a compact, easy to understand, and mostly graphical description of a domain or system of interest. However, a simple but significant problem stands in the way of this goal—the UML's instantiation model does not scale-up cleanly to multiple modeling levels. As long as there are only two levels—one with classes (M_1) instantiating another level with objects (M_0), the UML instantiation mechanism works well. However, when the UML concepts are employed to describe the level from which the class level is instantiated, i.e., the UML metamodel level (M_2), they do not support a natural modeling approach. In particular, they fail to adequately describe an M_1-level model element that represents a class (for further instantiation) as well as just an object.

Unfortunately, the metamodeling approaches currently used or proposed for the UML are based on an instantiation mechanism which treats an instance of a model element as just an object. Although an instance can also represent a class[1], the properties that it receives by virtue of the instantiation mechanism are only those of an object. Thus, for instance, when an M_1-level element is instantiated from an M_2-level element all of the attributes and associations defined for the M_2-level element becomes slots and links of the M_1-level element, and thus are not available for further instantiation. If an M_1-level element wishes to have attributes and/or associations, these must either be defined explicitly or must be inherited. The inability of the instantiation mechanism to carry information concerning attributes and associations across more than one level of instantiation is the source of pernicious problems in the currently proposed UML metamodeling frameworks.

In this paper we examine the nature of the most serious of these problems, which we refer to as the "ambiguous classification" and the "replication of concepts" problems, and discuss some of the attempts to compensate for them in existing metamodeling approaches. We then introduce a fundamentally different approach to metamodeling which rectifies these problems and provides a sound basis for a multi-level modeling architecture. This approach is based on the premise that the definition of modeling elements should inherently recognize the possibility of instantiation sequences that span more than two modeling levels, and should allow attributes and associations to be obtained purely through instantiation. Finally we describe the key concepts upon which such a mechanism is based and explain how it overcomes the identified problems.

2 Symptoms of Shallow Instantiation

The current UML instantiation model, which for the purposes of this paper we call "shallow instantiation", is based on the premise that a class can only define the semantics of its direct instances, and can have no effect on entities created by further instantiation steps. This view of instantiation, which of course makes sense in traditional two-level environments, leads to two fundamental problems when scaled up to multiple metalevels.

[1] In the UML instances of a metaclass that specializes Classifier are viewed as elements which can themselves be instantiated.

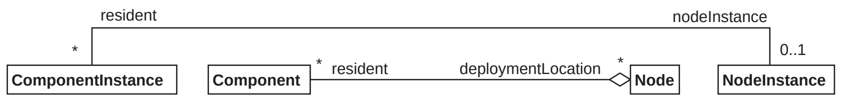

Fig. 1. Defining components and nodes

2.1 Ambiguous Classification

Traditionally, object-oriented modeling has only been concerned with two levels, the object (M_0) level and the class (M_1) level, where every element at the M_0-level is an instance-of[2] exactly one M_1 element. When a metamodeling level (M_2) is introduced, the situation is less clear-cut.

As an illustration of the problems that arise when this model of instantiation is scaled up to multiple levels, consider Fig. 1, which shows a simplified extract of the current UML metamodel related to the definition of nodes and components. These classes are regarded as residing at the M_2-level in the typical multi-level modeling architecture. In the UML, a component is an executable code module that can be deployed independently on physical computational resources known as nodes. A node is a run-time physical object that generally has at least a memory, and also often processing capability, upon which components may be deployed. The UML metamodel contains information about the nodes and component concepts in two places: The core package contains the M_2-level classes Node and Component and an association between them with role name resident for Component. The common behavior package of the metamodel, on the other hand, contains the M_2-level classes ComponentInstance and NodeInstance, and an association between them with a role name resident for ComponentInstance. This association enables component and node instances to be linked with each other. Note, however, that an additional constraint in the metamodel specifies that only component and node instances whose classifiers are linked with an resident/deploymentLocation relationship may be linked with each other. Figure 1 therefore illustrates how the UML metamodel captures the notions of nodes and components.

The problems of shallow instantiation reveal themselves when lower level classes are instantiated from the metamodel. Figure 2 shows a typical scenario containing M_1-level classes C and N, and M_0-level objects CI and NI. Class N is an instance-of the M_2-level element Node and since Node inherits from Classifier, class N can be used to instantiate an actual node instance NI. Similarly, class C is an instance-of the M_2-element Component and since Component inherits from Classifier, class C can be used to instantiate an actual node instance CI. One of the fundamental properties of component instances and node instances is that the former reside on the latter in a running version of the system. Therefore, one would like to define at the metamodel (M_2) level the fact that component instances, i.e., instances of ComponentInstance (at the M_0-level) have links to the

[2] As in the UML, specification, we use "instance-of" to refer to direct instantiation only, and "indirect instance-of" to refer to the relationship between an instance and one of the superclasses of its class.

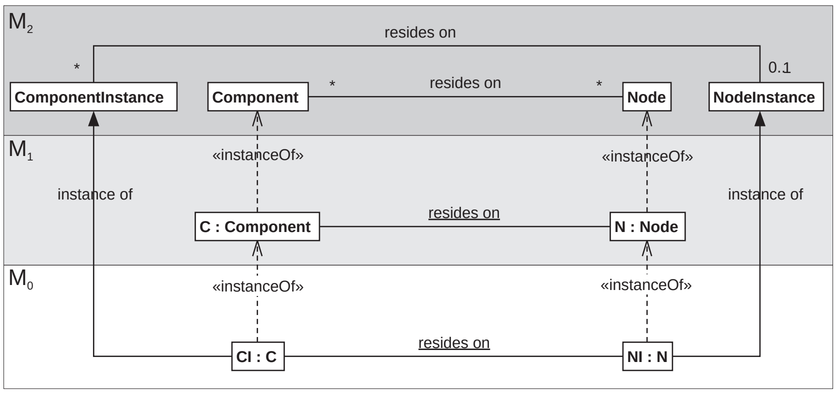

Fig. 2. Using the metamodel to define component and node instances

nodes instances (at the M_0-level) that they reside on. Not making such a statement at M_2 puts the responsibility on the modeler who creates a component type at M_1 to establish an association to a corresponding node type. Simply defining an association between Component and Node at the M_2-level, called, e.g., resides on, however, will not work because of the limitations of shallow instantiation. The problem is that when an instance of Node is created such as N, the association resides on becomes a link to an instance of Component such as C. This is highlighted in Fig. 2 by underlining its name and is actually desired since it specifies which component type may reside on a particular node type. Because this is a link, however, it can have no effect on actual node instances and component instances instantiated from N and C.

The only way in which the actual M_0-level component instances and node instances are able to obtain a link to capture which component instance resides on which node instance is by instantiating an association. This is why NodeInstance and ComponentInstance—as M_2-level representatives for "typical" component and node instances—feature the resides on association, supporting resides on links between node and component instances at the M_0-level.

This strategy of introducing representative types at the M_2-level is the adopted practice not only in the current version of the UML, but also in more recent proposals for the UML metamodel [2]. However, this approach clearly raises a dilemma—are objects at the M_0-level (e.g., NI) instances of their M_1-level classifiers (e.g., N), or instances of the representative instance at the M_2-level (e.g., NodeInstance), or both? From a modeling perspective a node instance is created by a class of type Node, as opposed to a (meta-)class called NodeInstance. In terms of the example used in the MML approach [3], the same dilemma occurs with regard to whether the object fido is an instance-of class Dog or of (meta-)class Object. In this example, one faces the need to express that fido is classified by Dog but also should be recognized as an Object (see the lower part of Fig. 3). In [3], this "ambiguous classification" problem is labeled the "multiple of" problem, because there are potentially two valid classifiers for one instance at the M_0-level.

In effect one of the instance-of relationships is serving as the *logical* instantiation link, while the other is serving as the *physical* instantiation link. Interestingly, for NI the relationship which is naturally viewed as the primary one from the perspective of the modeler (the logical one to N) is not the same as the relationship which is naturally viewed as the primary one from the perspective of a CASE tool (the physical one to NodeInstance).

In essence, the multiple classification problem arises because one wants to make statements at the M_2-level about how node and component instances (at the M_0-level) are shaped and how they may be connected with each other. Yet, one is prohibited from doing so in a natural way by the semantics of shallow instantiation.

2.2 Replication of Concepts

The ambiguous classification problem arises because the traditional semantics of instantiation prohibits a model element from influencing anything other than its immediate instances. Had we been able to specify an association resides on between Component and Node at level M_2 with the ability to create an effect on M_0 elements, there would have been no need for the ComponentInstance and NodeInstance elements.

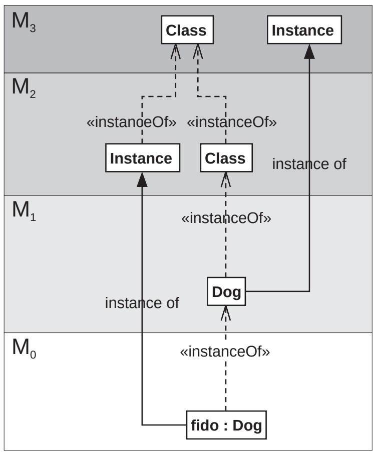

Fig. 3. Replication of concepts

Because shallow instantiation fails to carry information across more than one instantiation link it is often necessary to duplicate information at multiple levels. This is known as the "replication of concepts" problem. As an illustration of this problem, consider how the "fido the dog" problem mentioned in the previous subsection would appear when all metamodeling levels are considered. Fig. 3 shows the typical scenario in which fido is an instance-of Dog , Dog is an instance-of Class and Class (at the M_2-level) is an instance-of Class (at the M_3-level).

For the reasons explained in the previous subsection, with shallow instantiation model elements usually have more than one instance-of relationship in order to obtain additional "auxiliary" properties. In the case of the UML, all model elements instantiated from a class are also viewed as being instances of the metaclass Instance, or one of its subclasses. This is illustrated in Fig. 3. Identifying fido to be an Instance as well as a Dog can also be interpreted as defining the semantics of instantiation. This is the approach to the definition of semantics adopted in the MML work [3].

The same pattern is repeated again at the level above to define the properties exhibited by the model element Dog. The relationship between Dog and its classifier Class now represents a logical instance-of, and its real physical classifier is now considered to be Instance (this time at the M_3-level). In effect, the semantics of instantiation have to be redefined at each level (other than the lowest level).

The replication of model elements and concepts at multiple levels causes two main problems, First, it means that the size of the models are increased, making them more complex and difficult to understand. This phenomenon can be seen in the MOF and the UML, where a large proportion of the model elements and concepts are the same. Second, when fundamental concepts such the instance-of relationship are redefined and recreated for every level, inconsistencies can easily arise, leading to subtle differences in instantiation semantics at different levels.

3 Compensating for Shallow Instantiation

The problems described above have not gone unnoticed by the research community, and several strategies have been developed to try to cure the symptoms. We review these briefly below.

3.1 PowerTypes

One way of relieving the symptoms of shallow instantiation is to use the inheritance mechanism as well as instantiation to capture instance-of relationships. One strategy for achieving this is the powertype concept [4]. It avoids the presence of two direct instance-of relationships for one element by making one of them indirect.

The way in which the powertype approach addresses the ambiguous classification problem (illustrated in Fig. 2) by use of polymorphism is shown in Fig. 4. As can be seen, NI is a defined to be a direct instance-of N and an indirect instance-of NodeInstance

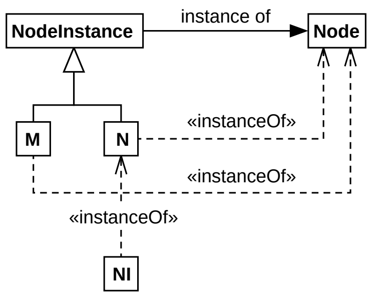

Fig. 4. Node as a powertype of NodeInstance

Object NI obtains its features from NodeInstance not by being a *direct* instance-of it (see Fig. 2) but by being an *indirect* instance-of it. As N receives all features from NodeInstance by inheritance, it can shape NI as necessary. In this case an association resides on from NodeInstance would enable NI to be linked to component instances.

In general, a powertype is defined to be a type (e.g., Node) whose instances (e.g., N and M) are subtypes of another type[3] (e.g., NodeInstance). In this example, therefore, the powertype of NodeInstance is Node. Note that also associations involving Node, such as resides on), can have an effect on N because Node is viewed as its classifier.

The important contribution of the powertype approach is to exploit the fact that NI does not need to be a direct instance-of NodeInstance but that it suffices

[3] Type, here, can be taken to be synonymous with classifier.

to model it as an indirect instance. More generally, the powertype concept recognizes that one sometimes needs to characterize a concept as being an instance-of a special type *and* to be subtype of another special type, as opposed to an "either/or" choice.

However, the powertype concept, as is, does not fit within the strict four layer modeling architecture of the UML. Although Node is assumed to be the classifier of N it resides at the same level. In other words, Node would have to be moved to level M_2 in order to exploit the powertype concept within the UML. This is, in effect, done by the approach described in the following section.

3.2 Prototyipcal Concept Pattern

Like the powertype approach, the Prototypical Concept Pattern [5] solves the ambiguous classification problem by combining the inheritance and instantiation mechanisms, but does so in a way that is compatible with ordinary shallow instantiation semantics. Instead of requiring a new relationship type to express that Node is the powertype of NodeInstance, Node is modeled as the classifier of NodeInstance. Figure 5 shows how to apply the Prototypical Concept Pattern to classify NI as both being an instance-of N and an indirect instance-of NodeInstance.

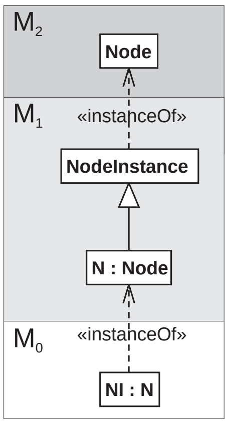

Fig. 5. The Prototypical Concept approach

The instance NI is, of course, an instance-of its class N (see Fig. 5). It is, however, also an indirect instance of NodeInstance because N inherits from NodeInstance. An association to ComponentInstance defined for NodeInstance, therefore, would create a link in NI to a component instance e.g., CI.

Figure 6 shows the complete scenario for the component/node model. Note how N receives the required resides on link (with the is allowed to reside on interpretation) by virtue of being an instance-of Node. The idea of a prototypical concept, which conveys certain properties upon all its subtypes has a long tradition in programming languages. The root class Object is used in the libraries of such languages as SMALLTALK, EIFFEL, and JAVA.

The potential problem with this approach is that the inheritance link between a new class (e.g., N) and its prototypical concept (e.g., NodeInstance) relies on a convention. There is no means available within the UML metamodel to enforce its presence. Moreover, this approach does not solve the replication of concepts problem.

3.3 Nested Metalevels with Context Sensitive Queries

The two previous approaches tackle the problem of ambiguous classification by using specialization to handle one of the classification dimensions. They also as-

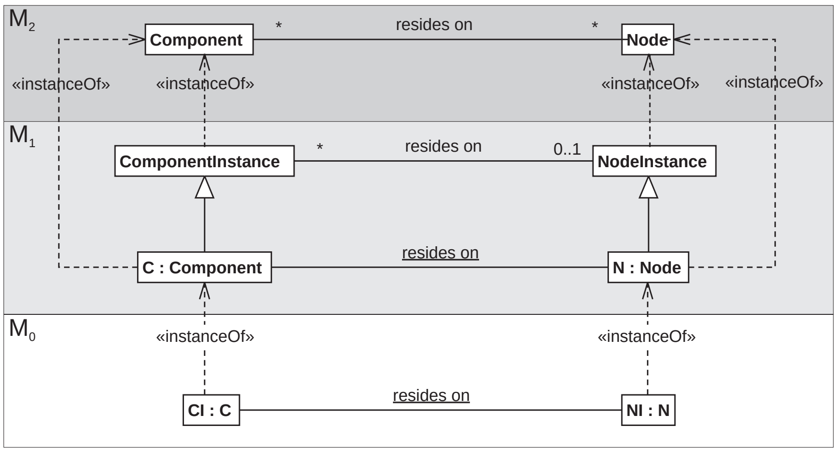

Fig. 6. Applying the Prototypical Concept Pattern

sume either a flat modeling framework (powertypes) or the linear metamodeling framework that currently underpins the UML (prototypical concept pattern).

One way of tackling the symptoms of shallow metamodeling in a way which does not rely on specialization is to organize the level in a more sophisticated way than just a linear hierarchy. In the nested metalevels approach of [3] one assumes that a level M_x which is meta to both M_{x-1} and M_{x-2} sees no level boundary between M_{x-1} and M_{x-2}.

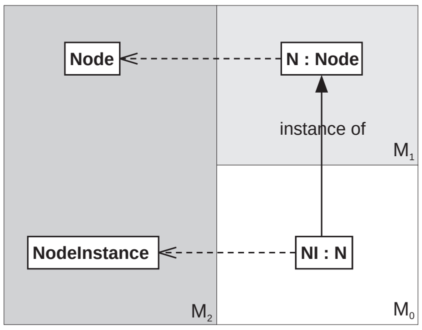

Fig. 7. Nested metalevels

This is depicted by Fig. 7. From the perspective of level M_2, N is an instance-of Node and NI is an instance-of NodeInstance. The level boundary between M_1 and M_0 is non-existent for level M_2. This view is justified by the fact that not all instance-of relationships are of the same kind [6]. The ambiguous classification problem is resolved by giving the physical instance-of relationships between Node and N, and NodeInstance and NI first class status, while the "instance of" relationship between N and NI are viewed as only being links between instances. Although the instance-of relationship between NI and NodeInstance would be non-strict [7] under normal circumstances, it is not here because levels M_0 and M_1 are assumed to be a single level from the perspective of level M_2.

The drawback of this approach, however, is that the instance-of relationship between NI and N has second-class status. In the "fido the dog" example fido would be an instance-of Object and not of Dog. This is not only counterintuitive for someone focusing on the levels M_1 and M_0, but also implies that this second-class instance-of relationship has to be manually created between NI and N by some M_2-level mechanism.

Since a user not concerned with metamodeling would prefer to view NI as an instance-of N (or fido as an instance-of Dog) Álvarez et al. propose to make queries concerning classifiers context dependent [3]. Asking NI for its class would therefore yield N at the modeling level and NodeInstance at the metamodeling level. To achieve an adequate representation of the model for both kinds of users, Álvarez et al. propose a mapping G, which maps models between meta-levels and transforms them accordingly.

While this approach sidesteps the multiple classification problem and is probably faithful to how model representations of current case-tool vendors are designed, it still clings to the instantiation semantics of the traditional two level (M_1 and M_0) modeling approaches. As a consequence, the class features of a model element still have to be defined manually at each level. For instance, there is no way for the class Node to automatically influence the model element NI. Thus, the representative instance (introducing NodeInstance) approach is still required. Moreover, the nested metalevels approach does not provide an answer to the "replication of concepts" problem.

4 Deep Instantiation

Although they go someway to alleviating the symptoms of shallow instantiation semantics, none of the approaches described in the previous section satisfactorily solve both the "multiple classification" and "replication of concepts" problems. Moreover, they all involve uncomfortable levels of complexity. We believe that these approaches only partially relieve the symptoms, without curing the underlying problem. As mentioned previously, the underlying cause of the symptoms is the reliance on the old "two-levels only" modeling philosophy, i.e., the shallow instantiation assumption. For this reason we believe the solution lies in a more fundamental enhancement of existing modeling frameworks.

It is well known that model elements in a multiple level framework can represent both objects and types. This is in fact the basis of the very concept of metamodeling. Odell [4] captures this property by stating that objects are also types, while Atkinson uses the term "clabject" [7] to refer to a modeling element with both class and object facets. The capability of instances to be types is also inherent in the definition of the MOF [8].

Although the dual object/class properties of many modeling elements is widely acknowledged, it is not adequately supported by the traditional shallow instantiation model. On the contrary, the traditional instantiation model is "shallow" precisely because the class facet of a model element always has to be explicitly documented for each model element that represents a type. In other words, a class can never receive attributes and associations from its classifier, only slots and links.

We believe the solution to the problems outlined in the previous sections is to define an instantiation mechanism that recognizes and fully supports the class/object duality of instantiatable modeling elements within a multi-level modeling framework. In other words, an instantiation mechanism is needed in which a modeling element's class features can be acquired automatically by the instantiation step rather than always having to be defined explicitly. We refer to such an instantiation mechanism as deep instantiation. Only with a deep instantiation approach will it be possible to coherently define the instance and class facets of elements once and only once.

4.1 Potency

The key to achieving deep instantiation is to add the concept of "potency" to every model element at every level in a modeling framework. The potency of a model element is an integer that defines the depth to which a model element can be instantiated. Thus a model element of potency 0 corresponds to an object, a slot, a link, or any concept that is not intended for further instantiation (e.g., an interface or an abstract class). On the other hand, an element of potency 1 corresponds to a class, an attribute or an association that is intended to be instantiated only once. By extension, an element of potency 2 can support instantiation twice. In other words, it gives rise to instances with a potency of 1. The act of instantiating a modeling element obviously reduces its potency by one. This approach therefore unifies the concepts of class and object into a single concept "modeling element", distinguishing types (classes) and instances (objects) by their respective potency.

Another important property possessed by every model element is its level. As its name implies, the level of a model element is an integer representing the model level in which the element resides. Thus, an M_0 element has level value 0, and an M_1 element has value 1, etc.

Together, the level and potency properties of model elements allow many of the properties of the desired instantiation mechanism to be expressed in a very concise and concrete way. Instantiation, within a strict multi-level modeling framework, takes a model element with level l ($l > 0$) and potency p ($p > 0$) and yields an element with level $l - 1$ and potency $p - 1$. Since instantiation reduces both values by 1, it can only be applied to model elements whose potency and level are greater than 0. More formally,

$$\forall m \in \mathit{ModelingElement} : m.l > 0 \land m.p > 0 \Rightarrow \\ n.l = m.l - 1 \land n.p = m.p - 1, \text{where } n = @^4 m.$$

Elements whose potency is 0 cannot be instantiated, regardless of their level number. The potency property of a model element, hence, subsumes the isAbstract attribute (defined for GeneralizableElement in the UML) which designates a model element as being uninstantiatable. A model element which has an isAbstract slot containing the value *true* has a potency of 0. Another good example of a model element with potency 0 is the inheritance relationship between two elements, as it can not be further instantiated.

[4] Using an instantiation notation adopted in MML.

Many characteristics of the meta-modeling architectures in use today, such as the OMG's MOF and UML architectures can be concisely captured in terms of constraints on, and relationships between, the level and potency values. The fact that M_0 is assumed to be the bottom level for all elements, is captured by the rule that potency of a model element cannot be greater that its level, i.e,

$$\forall m \in \mathit{ModelingElement} : m.p \leq m.l$$

The fact that a modeling architecture only recognizes a certain number of levels is captured by a simple bound on the value of level, i.e.,

$$\forall m \in \mathit{ModelingElement} : 0 \leq m.l < 4$$

Finally, the semantics of shallow instantiation could be captured by the constraint that the potency of elements cannot be greater than one.

4.2 Single and Dual Fields

The semantics of instantiation is intimately related to the properties of attributes and slots[5]. When an element x is instantiated to create an element y (i.e., $y = @x$), the semantics of instantiation dictates that every attribute of x becomes a slot of y, with the same name and a value of the appropriate type. However, any *slots* in x have no effect on y, which we previously identified as the source of numerous problems.

Together, the potency and level values of a model element allow numerous simplifications of multi-level modeling. One of the most significant is the unification of the attribute and slot concepts. A slot is just an attribute that happens to have potency 0. In order to provide a convenient way of talking about attributes and slots in a general (unified) sense, and to avoid confusion arising from the semantic baggage associated with the existing terms, we introduce the concept of a field. A field with potency 0 represents a slot, and therefore must have a value. However, a field of higher potency may or may not have a value, depending on whether it is a simple field or a dual field.

A simple field is a field which does not have a value unless it has potency 0 (and thus corresponds to a traditional slot). Thus when an element x is instantiated to create an element y, any simple fields (of potency 2 or above) become an identical simple field of y, but with a potency decremented by one. On the other hand, a field of x with potency 1 becomes a field of y with potency 0 and a value.

A dual field is a field which has a value even for potencies greater than zero. When an element x is instantiated to create an element y, any dual fields of x become identical dual fields of y, but with a potency decremented by one and

[5] Methods types and their instances, of course, are also affected by the semantics of instantiation, but since they are basically the same as for attributes and slots they are not discussed further here.

with a possibly new value. A dual field therefore has no counterpart in traditional modeling approaches. The semantics of a dual field can best be thought of in terms of a set of simple fields. Basically, a dual field of potency n corresponds to a set of n simple fields, all identical except that they all have different potencies $(n, n-1, \ldots, 0)$, and the field of potency 0 has a value. The instantiation of an element with such a set of simple fields will yield a model element with $n-1$ similar fields. This is because the field with potency 0 cannot by definition be instantiated and thus disappears when the element it belongs to is instantiated.

Because they are model elements in their own right, fields also have their own level and potency values. Obviously the level of a field must match that of the model element to which it belongs, but the potency does not have to match. Clearly it makes sense to permit a model element to have fields with a lower potency, since these will simply disappear after the appropriate number of instantiation steps. However, perhaps surprisingly, it also makes sense for a model element with potency n to have fields with a potency higher than n. For instance, a regular abstract class corresponds to a model element of potency 0 with fields of potency 1. These fields cannot be instantiated, however, until a model element with a potency greater than zero inherits the fields.

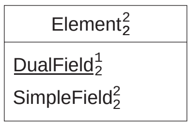

Fig. 8. Elements with potencies and levels

Clearly, providing a clean and consistent notation for these concepts is essential for its ultimate acceptance. At present the notation is at an early stage of evolution, but for the purposes of this paper we adopt the convention illustrated below. The potency of a model element is represented as a superscript and the level as a subscript. In addition, fields of potency 0 (which have a value) are distinguished by being underlined. We are aware of the fact that underlining is currently being used to denote static attributes but argue strongly for correcting this decision. Our potency 0 fields *exactly* correspond to instantiated attributes, whereas the correspondence between static attributes and instantiated attributes is only partial. Thus simple fields are underlined if their potency is 0, whereas dual fields are always underlined. The visualization of the level subscripts is optional, if clear from the context. Also, with one exception, it is only mandatory to explicitly indicate potency values greater than one. The exception are dual fields of potency 1 where the potency value has to be shown in order to distinguish the field from an ordinary field of potency 0.

4.3 Modeling with Deep Instantiation

In this section we demonstrate how the few simple concepts introduced in the previous section have the power to radically change the way in which metamodeling is performed, and how they provide a much simpler and conceptually coherent solution to the modeling scenarios introduced in the previous sections.

Fig. 9 shows a new version of the component/node scenario modeled using deep instantiation semantics rather than shallow instantiation. In order to illustrate how the concepts introduced in the previous subsection help simplify the presentation of components, nodes and their properties we have added a few

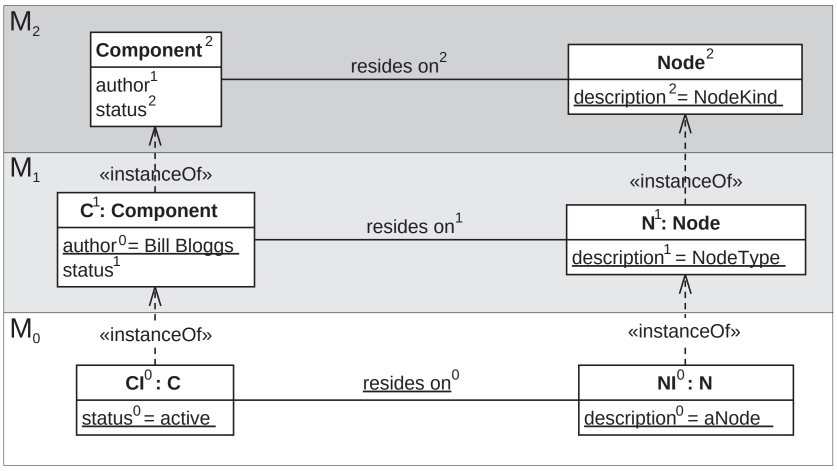

Fig. 9. Components and nodes with deep instantiation

additional concepts not present in the UML metamodel. However, the intention of these should be clear.

Field author is a potency 1 field of Component which becomes a potency 0 field in C and obtains a value. This, therefore, represents the normal attribute/slot semantics familiar in regular two-level modeling. Status, however, is a potency 2 simple field of Component. When C is instantiated from Component it becomes a potency 1 field (i.e. a regular attribute). Thus, when CI is instantiated from C, status becomes a potency 0 field (i.e. a regular slot) and assumes a value.

The description field of Node is a dual field rather than a simple field. This means that even though it has potency 2 in Node it also has a value. When N is instantiated from Node, the potency of description is reduced by one, but since it is a dual field it still has a value. Finally, when NI is instantiated from N, description becomes a potency 0 field with a value. There is consequently no difference between dual fields of potency 0 and simple fields of potency 0.

The problem of the proliferation of elements, associations, and links is also neatly solved by potencies. In Fig. 9 the resides on relationship between Component and Node has potency 2 rather than the traditional potency 1. This means that when C and N are instantiated from Component and Node, resides on becomes a relationship of potency 1. As a consequence it is available for further instantiation, and thus can give rise to potency 0 relationships (i.e. links) between instances of C and N. The need to introduce artificial metaclasses such as Component Instance and Node Instance, is therefore avoided. Note that the M_1-level resides on association precisely specifies which component types may reside on a particular node type. There is no need to allow links between component and node instances in general, which have to be restricted by an additional rule to achieve the effect that deep instantiation naturally provides. Also, in contrast with the prototypical concept approach, all properties associated with component instances and node instances can be defined simply and directly at the M_2-level in a way that is enforceable and controllable by tool vendors.

4.4 Metamodel for Multiple Metalevels

It has long been the goal of the graphical modeling community to capture all the fundamental ideas for modeling in a multilevel framework within a small, coherent and simple core, usually termed a meta-metamodel. So far, however, this vision has been frustrated by the problems described in the first section of this paper. For instance, the MOF, which is supposed to be the meta-metamodel for the UML, duplicates many of the same modeling concepts as the UML (because of the replication of concepts problem) and equivocates on the proper positioning of modeling elements [8].

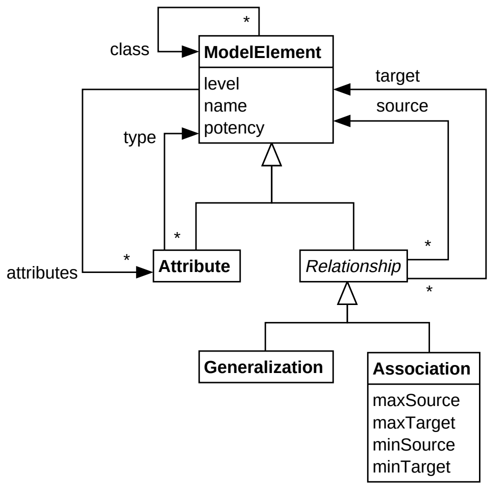

Fig. 10. The MoMM

We believe the concepts described above provide the ingredients for this vision to finally be fulfilled, and for a truly concise meta-metamodel of strict multilevel modeling to be defined. A preliminary version of the MoMM (**M**etamodel f**o**r **M**ultiple **M**etalevels) is illustrated in Fig. 10.

The key point to note about this meta-metamodel is that since every model element, anywhere in the metamodeling architecture, is intended to have level and potency values, these have to represented as dual fields of the basic model element. They must be dual fields because all elements in an instantiation chain, up to a maximum of three, need a level and potency value.

5 Conclusion

Given the OMG's goal to make the next version (2.0) of the UML supportive of a family of languages, the nature of the metamodeling framework will be critical in its future success. As a consequence, there has been a significant increase in the level of research into the optimal modeling architecture. However, all existing approaches proposed to date suffer from the limitations imposed by the model of shallow instantiation inherited from the traditional two-level roots of object technology. As a consequence they all suffer from a number of fundamental problems that seriously undermine the semantics of the metamodeling architecture.

The basic problem is that the traditional two-level semantics of instantiation does not scale up for metamodeling hierarchies with more than two levels. The original UML modeling concepts are sufficient to allow modeling at the M_1-level, but fail to address the needs at level M_2. This problem causes a number of symptoms in existing multi-level modeling architectures, such as the UML, including the “ambiguous classification” problem (or multiple-of problem as it is known in the MML work), and the “replication of concepts” problem.

Although various strategies have been developed for softening the impact of these problem, none is entirely successful in removing all of the symptoms. In this paper we have therefore put forward a number of principles that we believe offer a fundamental solution to these problems, and thus offer the way for a clean, simple and coherent semantics for metamodeling. We have demonstrated how these ideas can simplify the descriptions of real modeling scenarios currently found in the UML. Furthermore, the principles lead to a simple, linear hierarchy in which every modeling element can be assigned its proper location, i.e., the doctrine of strict metamodeling is supported.

This work is at a preliminary stage, and is currently ongoing at the Component Engineering group (AGCE) at the University of Kaiserslautern. However, we hope the ideas might be helpful in the search for the optimal UML 2.0 modeling framework.

References

1. Brian Henderson-Sellers. Some problems with the UML V1.3 metamodel. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. Institute of Electrical and Electronics Engineers, 2001.
2. Tony Clark, Andy Evans, Stuart Kent, Steve Brodsky, and Steve Cook. A feasibility study in rearchitecting UML as a family of languages using a precise OO meta-modeling approach. http://www.cs.york.ac.uk/puml/mml/mmf.pdf, September 2000.
3. José Álvarez, Andy Evans, and Paul Sammut. MML and the metamodel architecture. Workshop on Transformations in UML (WTUML'01), associated with the fourth European Joint Conference on Theory and Practice of Software (ETAPS'01), Genova, Italy, January 2001.
4. Jim Odell. Power types. *Journal of Object-Oriented Programming*, May 1994.
5. Colin Atkinson and Thomas Kühne. Processes and products in a multi-level metamodeling architecture. *submitted for publication*, 2001.
6. Jean Bézivin and Richard Lemesle. Ontology-based layered semantics for precise OA&D modeling. In Haim Kilov and Bernhard Rumpe, editors, *Proceedings of the ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques*, pages 31–37. Technische Universität München, TUM-I9725, 1997.
7. Colin Atkinson. Meta-modeling for distributed object environments. In *Enterprise Distributed Object Computing*, pages 90–101. IEEE Computer Society, October 1997.
8. OMG. Meta object facility (MOF) specification. OMG document formal/00-04-03, Version 1.3, March 2000.