Karlsruhe Reports in Informatics 2016,11
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Deep Modeling through Structural Decomposition

Georg Hinkel

2016

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Deep Modeling through Structural Decomposition

Georg Hinkel
FZI Research Center of Information Technologies
Haid-und-Neu-Straße 10-14
76131 Karlsruhe, Germany
hinkel@fzi.de

ABSTRACT

In some applications, traditional metamodeling in two levels gets to its limits when model elements of a domain should be described as instances of other model elements. In architecture description languages, components may be instances of their component types. Although workarounds exist, these require many validation constraints and imply a cumbersome interface. To obtain more elegant metamodels that require less constraints, deep modeling seeks ways to represent non-transitive instantiation chains. However, these concepts often make existing techniques for model transformation and analysis obsolete as these languages have to be adapted. In this paper, we present an approach to realize deep modeling only through structural decomposition, which can be implemented as a non-invasive extension to meta-metamodels similar to Ecore. As a consequence, existing tools need not be adapted. We validate our concept by creating a deep modeling architecture description language and demonstrate its advantages by modeling a synthetic web application.

CCS Concepts

•**Software and its engineering** → **Model-driven software engineering; Architecture description languages;** *Abstraction, modeling and modularity;* •**Computer systems organization** → n-tier architectures;

Keywords

Deep Modeling; Structural Decomposition; Refinements; NMeta; Modeling Language;

1. INTRODUCTION

When working with model-based or model-centered approaches such as Model Driven Engineering (MDE), mod-els always conform to a metamodel defining their structure. Since the metamodel itself conforms to a meta-metamodel and therefore the structure of the models is well-defined, models conforming to a metamodel can be automatically processed by model transformations to other artifacts such as code, documentation, or other models. As a consequence, any complexity in the metamodel also implies a higher complexity in dependent artifacts such as model transformations. It is therefore desirable to keep *metamodels as simple as possible* in order to minimize accidental complexity.

However, many of the most often used modeling formalisms such as Ecore usually do not allow a direct representation of the relation that a model element is an instance of another, except that all model elements are instances of their class [1]. This leads to accidental complexity, as metamodels must be aided by helper constructs. Instantiation relations must be described using references, for example using some kind of connector classes. The semantics of instantiation is then restored by introducing OCL constraints that ensure a valid usage to mimic instantiation relations. While this solves the modeling problem in the first place, it makes automated tool support difficult as such tool support has to analyze the OCL constraints to reconstruct the original intention.

At the same time, tool support is one of the main reasons that hinders MDE from a wider acceptance in the industry, as studies from Mohaghegi and Staron suggest [2], [3].

Approaches that aim to directly support instantiation relations between model elements are referred to as Deep Modeling or Multi-Level Modeling concepts. The latter term originates from the idea that such approaches not only support the usual two levels of metamodels and models, but that a model repository may contain non-transitive instantiation chains of arbitrary length. This means that a model element A can be an instance of another model element B which is an instance of model element C. However, unlike inheritance, instantiation is not transitive so that A is not an instance of C.

Most existing approaches to describe these instantiation chains use radically new concepts to describe these instantiations and thus yield the risk that developers may not use them but try to stay with the technologies they are used to as far as they can. Such mentality was shown for general purpose languages by Meyerovich [4], and we suspect that this is also true for modeling languages.

A further potential disadvantage is that all the subsequent tools such as model transformations have to be adjusted for Deep Modeling, as e.g. done by Atkinson [5] with an adjusted version of ATL called DeepATL. Given the plethora of model transformation languages, where even the most commonly used ones have much smaller user bases than most general purpose languages, we think that few transformation languages will be adopted and maintained for Deep Model-

ing.

In this paper, we propose a pragmatic approach how Deep Modeling, i.e., instantiation chains of arbitrary length, can be realized using only two *non-invasive extensions* to meta-metamodels aligned with EMOF, such as Ecore. The crucial advantage of this approach is that all the tools available for such a meta-metamodel can be reused and existing metamodels do not have to be changed. Thus, Deep Modeling can be introduced in a stepwise evolution process and only where it is beneficial.

The rest of the paper is structured also in multiple levels. In a first level, we present our notion of constraints and refinements in Section 2, explain how we use it for Deep Modeling in Section 3 and present our implementation in Section 4. As an instance of these concepts, we apply them to create a deep modeling architecture description language based on the Palladio Component Model [6] in Section 5. We instantiate this language for validation to model a simple web application serving media files in Section 6. Finally, we show related work in Section 7 and conclude the paper in Section 8.

2. STRUCTURAL DECOMPOSITION

In this section, we formally define our notion of structural decomposition that we use in this paper.

In a metamodel, the structural properties of a metaclass are determined by attributes and references, in Ecore referred to as structural features. The goal of our structural decomposition approach is to be able to decompose this structure as we specialize the metaclasses.

For example, consider a metamodel of vehicles. Vehicles may have wheels. However, for a car we can actually decompose this set of wheels into front wheels and rear wheels, and know more about the types of wheels that can be put to a car. If we tried to take a wheel off a bicycle and were to put it to a car, we would actually want the modelling environment to tell us that this is not possible, as soon as possible.

To describe this effect, we use a formal syntax close to the OCL standard. Types, denoted with capital letters, are ordered with a partial order relation $\preceq$ describing inheritance, i.e. $A \preceq B$ if B is an anchestor of A. The instances of a type A are denoted as $I(A)$. We denote collections of type A with its Kleene closure $A*$.

A feature $f : A \to B$ of a class A is a pair of two functions: The getter function $f.get : I(A) \times \Omega \to I(B)$ maps a model element of type A to a type B (either also a metaclass or a primitive type), depending on the global state $\omega \in \Omega$. This state space Ω is adapted from stochastics and models that the model elements are mutable. The partial setter function $f.set : I(A) \times I(B) \times \Omega \rightharpoonup \Omega$ modifies the global model state to set the feature of a given model element. Its domain describes the set of valid model states. In particular, we say the feature f has a valid value for a model element a in state ω if $(a, f.get(a, \omega), \omega) \in \mathcal{D}(f.set)$. The getter and setter of a feature have to comply with a SetGet-law specifying that the getter should return what the setter stores. In particular, if $(a, b, \omega) \in \mathcal{D}(f.set)$, we have that

$$f.get(a, f.set(a, b, \omega)) = b. \tag{SetGet}$$

The SetGet-law is inspired by and closely related to the PutGet-law from Lenses [7], but is made for mutable model elements.

Definition 1 (Structural decomposition). *Let A and B be types. A set of features $f_1, \ldots, f_n : A \to B*$ for types A and B and an $n \in \mathbb{N}$ is a* structural decomposition *of a feature $f : A \to B*$ if we have that for each global state $\omega \in \Omega$ and $a \in A$ that*

$$f.get(a, \omega) = f_1.get(a, \omega); f_2.get(a, \omega); \ldots; f_n.get(a, \omega);.$$

We say that f is made of $f_1, \ldots, f_n$ and call the f_i components of a composition f.

Since there is an embedding from $A \times \Omega \to B$ into $A \times \Omega \to B*$, we will also allow the features used for decomposition to be single-valued where we depict an element `null` $\in B$ that corresponds to an empty string in $B*$. Likewise, we allow compositions to be single-valued. In this case, the value of the composition has to match the only component value that is not `null`.

We combine structural decomposition with a notion of refinement as per the following definition:

Definition 2 (Refinement). *Let A, B, C and D be types with $C \preceq A$ and $D \preceq B$. Further, let $f : A \to B$ and $g : C \to D$ be features. Then, we say that g is a refinement of f if $f.get$ and $g.get$ are the same on $I(C) \times \Omega$ and the setters are the same for elements of C, i.e. the following equations holds for all $c \in I(C)$, and $\omega \in \Omega$:*

$$f.get(c, \omega) = g.get(c, \omega)$$
$$\mathcal{D}(f.set) = \mathcal{D}(g.set) \cap I(C) \times I(B) \times \Omega \subset I(C) \times I(D) \times \Omega$$

and if $(c, b, \omega) \in \mathcal{D}(g.set)$, then we have that

$$f.set(c, b, \omega) = g.set(c, b, \omega).$$

In particular, we know that for each element c, the reference f will always (i.e. for each model state ω) refer to an element of D.

An important special case here is the refinement by a constant reference $g \equiv d$ for some constant element $d \in I(D)$. Usually, constant features are not explicitly modeled as they do not contain any information specific to the model element, but in combination with a refinement, they may carry information that is known for some subtypes, but not in the general case for a given type A.

3. DEEP MODELING THROUGH STRUCTURAL DECOMPOSITION

Throughout this paper, we treat as the main characteristic of Deep Modelling that non-transitive instantiation relationships can be modeled, i.e. there can be model elements A, B and C such that A is an instance of B and B is an instance of C but A is not an instance of C.

A prominent example depicted in Fig. 1 is that a concrete dog (a poodle) can be an instance of `Poodle` which in turn is an instance of `Breed`. In this example, `Poodle` acts both as model element (object) and as a class, which is why it is often called a *clabject* to express this duality [8].

Such a situation can be described using the powertype pattern first presented by Odell [9]. However, many modeling environments such as Ecore currently do not support this pattern and even in the UML, it is an isolated concept.

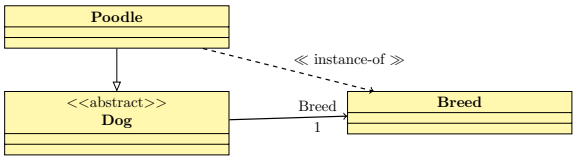

Figure 1: Poodle is both a class and an instance

On the other hand, there is a very prominent example of a clabject even in traditional two-level modeling with self-descriptive meta-metamodels, namely the class `Class` which is an instance of itself. Its properties as a class are described by the references and attributes of `Class` because the meta-metamodel effectively describes a type system. Essentially, the class `Class` describes that its instances can be instantiated, typically done through a mapping to a platform class that we call code generation.

The core idea of this paper is to reuse this duality of the `Class` element on a wider scope. Thus, whenever we conceptually face a clabject, a metaclass whose instances can be instantiated again, this metaclass should be a subtype of `Class`. However, unlike UML stereotypes that were rejected by a large portion of the community, we hide the type system relevant information by decomposing them into domain references. References that are not applicable are cut off using a refinement with a constant.

Furthermore, we added an explicit instance-of relation between classes. With this relation, we may specify that `Dog` is an instance of `Breed`, meaning that the type of each dog element will be in turn an instance of `Breed`. Thus, the `Breed` reference is just a converted type reference and hence, the instance-of relation can be seen as a formalization of the powertype pattern, expressed with a single reference.

Given the relationship to stereotypes, this connection is hardly suprising since Henderson and Gonzalez-Perez have already shown a close connection of the powertype pattern to UML stereotypes some years ago [10]. We will see the instance-of relation being used later in Section 5 in a number of places.

But as the domain concepts are still classes as they (possibly indirectly) inherit from the class `Class`, we can use the standard generators to generate model representation code for them. With this generated code, the class nature of a clabject is represented by a mapping into the platform type system while the object nature of it is represented as a model element.

This mapping of the type facet to the platform type system also yields the consequence that an instance of a clabject A cannot be in the same model as A unless the model is manually bootstrapped, such as done for `Class`. As a consequence, a clabject cannot easily be an instance of itself unless the model developer has explicitly expressed an intend that this behavior is desirable by bootstrapping the model. This neglects the various paradoxa presented by Atkinson et al. [11] for languages he referred to as level-blind.

On the other hand, the instantiation indeed implies a stratification of the model and divides it into levels, as suggested by Atkinson [11], though these levels can be crossed not only by instantiation relations. Our case study example in Section 5 will give an example where this enables us to keep a level structure in the presence of model elements that would otherwise break the level structure. Basically, this is possible by binding user defined clabjects to classes known in advance through the instance-of relation. These base classes are known before any model for the deep metamodel is created and thus can freely be referenced, including a usage for analysis or transformation purposes.

4. NMETA

We have implemented structural decomposition and refinements in the NMeta meta-metamodel which is used within the .NET Modeling Framework (NMF[1]). Its basic metaclasses are depicted in Figure 2. In NMeta, every model element is an instance of `ModelElement` and therefore has an absolute URI which makes it uniquely identifiable and addressable. This URI is created automatically based on the containment hierarchy of the model elements (the `Parent` reference) and on the URI of the model that contains the model element. Further, model elements also have a relative URI to identify them within the scope of their model in case the models URI is not set. Conversely, it is possible to resolve a relative URI starting from a given model element as context.

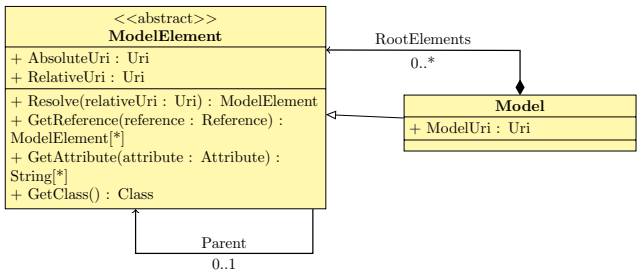

Figure 2: The base metaclasses in NMeta

Model elements are categorized into classes that define the type system of a model (metamodel). An excerpt showing the metaclasses responsible for the type system is depicted in Figure 3. The type system is similar to the one from Ecore and there is a model transformation from Ecore to NMeta. In particular, a model conforming to an Ecore metamodel M can also be read using the transformed NMeta metamodel M', provided M does not contain generic types, factories or custom XMI handlers.

Similar to Ecore, metaclasses may contain attributes and references, but attributes and references may refine other attributes or references and constraints may be put on them. If an attribute is refined or constrained, it is effectively structural decomposed by all the refining attributes and a constant attribute yielding the values of the constraint. An equivalent statement holds for references.

The reference refinement is used e.g. for the `Type` reference that attributes and references inherit from `TypedElement`. The reason is that a reference is only valid when the type assigned to it is a reference type. Conversely, an attribute must be typed with a value type. In Figure 3, this relation is denoted with a dotted arrow from the `ReferenceType` or `DataType` reference of references and attributes to the `Type` reference of `TypedElement`.

The semantic behind this assignment is that the `Type` of a reference is its `ReferenceType`. Conversely, if we set the `Type` of a reference, the setter internally sets the `ReferenceType`.

[1] http://nmf.codeplex.com/

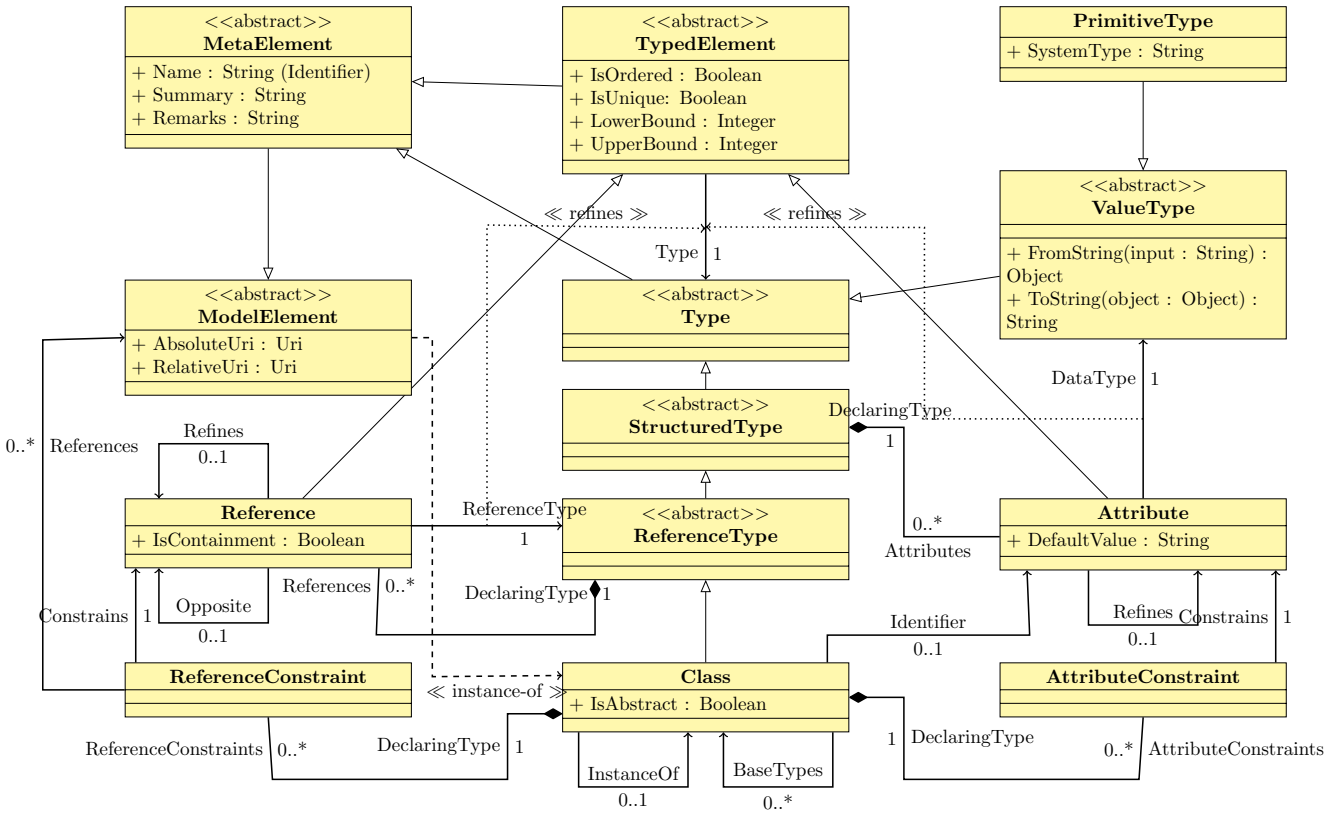

Figure 3: The type system of NMeta

However, this only works if the set value is an instance of `ReferenceType`. If it is not, we throw an exception because a reference whose type is not a reference type cannot be valid. This implementation yields that the validation that types of references must be reference types is already checked by the type system and thus there is no need for an additional constraint, formulated e.g. in OCL.

References may only refine references from base classes, but conversely it is allowed that a reference is decomposed multiple times for different subtypes of its declaring type. Components of a structural decomposed reference must match the composition reference in terms of ordering and containment and can themselves be decomposed again. A unique reference must not be decomposed as we did not implement to ensure a uniqueness constraint for the composition reference. The default value of an attribute is ignored if this attribute is decomposed. Since the inheritance hierarchy of value types is not modeled, the types of refinement attributes must match the refined attribute exactly.

Besides a reference of base classes, classes in NMeta are allowed to restrict inheritance to instances of a given class through the `InstanceOf` reference. This reference may only be specified for abstract classes. If a class A is an instance of class B, then only instances of B may inherit from A. Consider a an instance of type C which inherits from A. Because B was declared as an instance of A, C must be of type B. Thus, the type of a is an object of type B. Since this is known at compile-time, the generated code contains a refined method to obtain the model elements type of type B.

If the `InstanceOf` reference is left blank, this has the same effect as specifying that a class is an instance of `Class`. This is because the base class `ModelElement` is marked as an instance of `Class`. Moreover, when classes define an instance-of relation and one of its base class also specifies an instance-of relation, then the new instance-of class must be a subtype of the base class instance-of class. As an immediate consequence, all classes used in the instance-of reference must be subtypes of `Class`.

NMF provides a transformation from NMeta metamodels[2] to code. This code generator creates an interface and a default implementation class for each class in the metamodel. For any attribute or reference, a property and a change event is generated. If there is no structural decomposition defined, this property is backed by a field, otherwise a private getter and setter implementation is generated that composes or decomposes the property on the fly. For decomposition, the setter will assign the value to the first component property that fits. This affects the default implementation class, as any implementation class using a backing field for a decomposed property cannot be reused, but it does not affect the generated interface, so that the substitution principle is maintained for any analysis that consumes the model and just relies on the interface. This holds in particular also for the code generator itself, so that a model representation class can be generated for every subclass of `Class` as well. Artifacts that modify model elements such as editors would rather op-

[2]When using deep modeling approaches, the term metamodel gets blurred. We use the term metamodel to describe a model that can be instantiated.

erate on the real type of the model elements and therefore only see the public properties, which are exactly the non-decomposed properties.

The generator links a metaclass with a generated class through an annotation. Furthermore, the method to return the class element of a model element is overridden to return the concrete class element. If an abstract class A specifies an instance-of relation to clabject class B, then every instance of A will have a type that is itself of type B and therefore, an abstract discriminant method is generated that returns the type as an instance of B. For any instance of B, the code generator will generate an implementation of this method that statically returns this model element.

5. CASE STUDY: AN ARCHITECTURE DESCRIPTION LANGUAGE USING DEEP MODELING

To validate our concept for Deep Modeling, we created DeepADL as a prototype language for architectural description using Deep Modeling. DeepADL is largely inspired by the Palladio Component Model (PCM) [6] but only focuses on component repositories, software architectures and deployment. In particular, the performance-relevant elements of PCM such as service effect specifications are omitted. Many of these concepts do not benefit from Deep Modeling and are therefore not relevant for this paper. However, our approach allows us to simply copy them from a strict two-level metamodel of PCM into a deep modeling metamodel of DeepADL, making this approach attractive for refactorings for deep modeling.

To answer where to apply Deep Modeling, one has to look for concepts that are best described with an instance-of relationship, for example using the patterns of de Lara et al. [1]. In an architecture description scenario, we have identified several of these instantiation relationships. First of all, an architecture description introduces models of a component and its interfaces. These component types form an application-independent repository. Instances of these component types are then assembled to a systems architecture. Here, each instantiated component (`AssemblyContext` in PCM) has to be connected to components realizing the interfaces that are required by the components type. In the next step, the architecture is deployed to a set of resources, where each component must be mapped to a particular resource.

A challenge here are composite component types, component types that use a set of component instances to fulfill their functionality. The problem arises because these component types 'live' in the component type repository level but require elements from the system architecture level – the instances of other component types. This is forbidden by all approaches we know that use a fixed level structure ('level-adjuvant' languages).

Together with NMeta itself, this makes five modeling levels which we sketched in Figure 4. These levels are quite natural as they reflect the development process of a component-based system. At first, a component model is created (or selected from the many already existing ones). Then, component types are created, before they can be assembled to a component-based system which in the end can be deployed. As foreseen by Atkinson et al., they also match the stratification of the model into levels. However, through abstract base classes, we are able to cross these level boundaries.

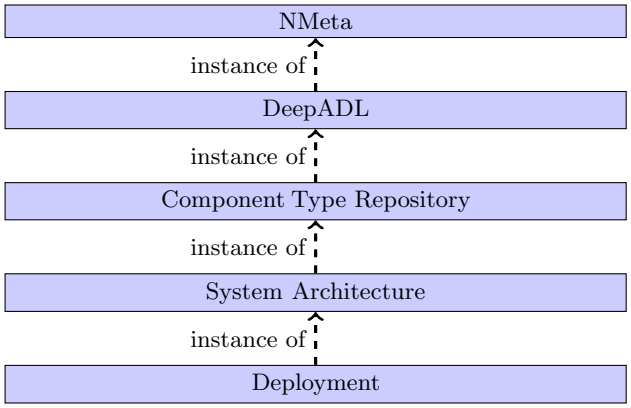

Figure 4: Levels of DeepADL

These levels also exist in PCM although it is modeled in Ecore which only supports strict metamodeling, i.e. exactly two levels. We will thus use it for comparison in Section 6.

We begin with component types. Component types are represented by the abstract class `Component`. Since component types can be instantiated, they are clabjects and therefore `Component` must inherit from `Class`. Same as PCM, DeepADL supports two different kinds of component types, basic components and composite components. Instances of a component type are assembly contexts, so we create a reference constraint to the base types of `Class` and fix it to the `AssemblyContext` class. Conversely, we specify that `AssemblyContext` is an instance of `Component`.

Further, a component type *is* an implementation of its provided interfaces. The most convenient way will be if the component types inherit from the interfaces that they provide. Thus, `Interface` itself must be clabject as well and the provided interfaces of a component type refine its base types. However, here we introduced a subtle difference to PCM since components in PCM are allowed to provide the same interface multiple times through multiple roles.

For each required interface of its component type, an assembly context must be assigned an assembly context whose component type provides the required interface. In PCM, this is modeled through an `AssemblyConnector`. In DeepADL, we want our assembly contexts to have a strongly typed reference for each required interface of the component type so that the ontological property of required interfaces of a component type becomes the linguistic element that the assembly context should have a reference. Hence, the required interfaces refine the references of `Component`. Thus, `RequiredInterface` must inherit from `Reference`. These required interfaces should not reference any reference type, but only instances of the class `Interface`, since any access to a component must be encoded through an interface.

Because component types are classes, we can map them to a CLR platform class and instantiate instances of component types. These instances are assembly contexts since any component type as a class also inherits from `AssemblyContext`. This makes the component type of an assembly a part of its identity since the type of an object cannot be changed during its lifecycle. Thus, unlike e.g. in PCM, the component type of an assembly cannot change and assembly contexts cannot exist without a component type. Here, important validation rules are ensured directly by the type system, which we see

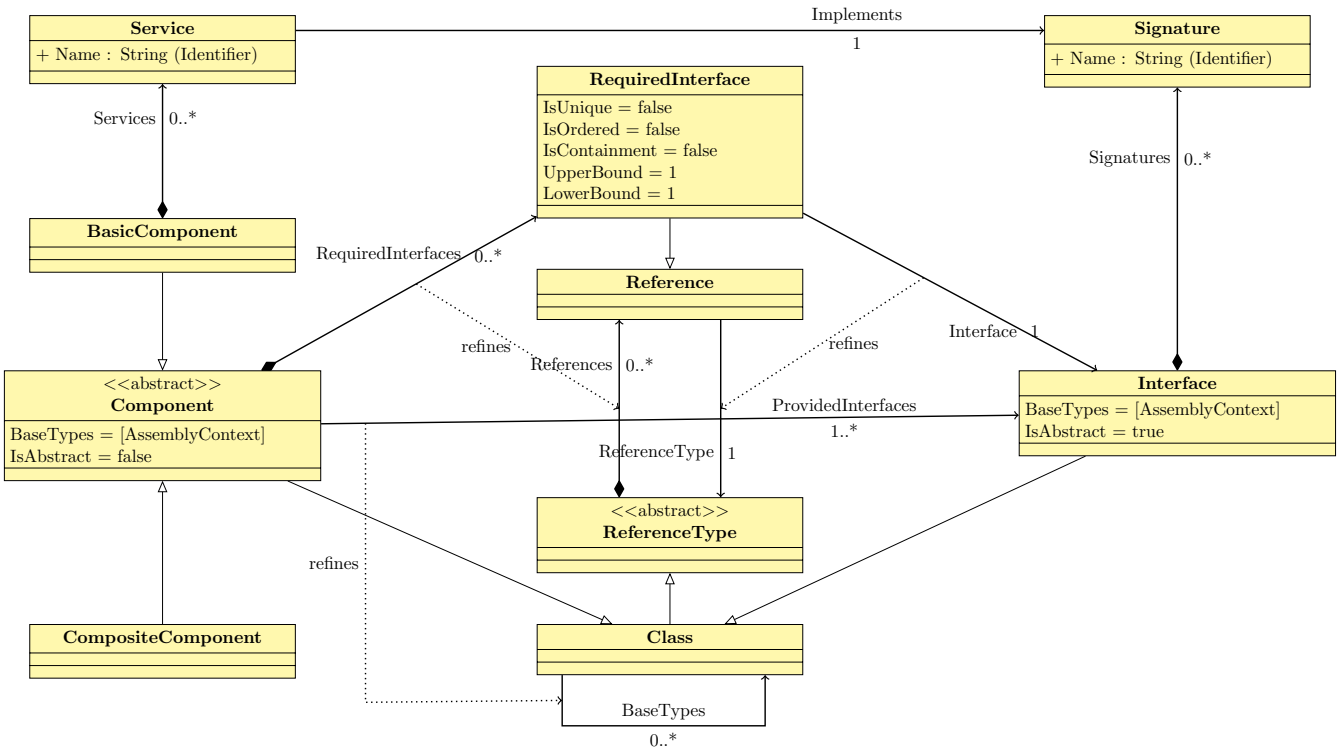

Figure 5: Component types in DeepADL

as a big advantage.

The signatures offered by an interface are not relevant at the system architecture level since the fact that a component type provides a particular interface already implies that a component provides services that implement the interfaces signatures. As a consequence, the ontological properties of an interfaces signatures or a component types offered services are no linguistic properties and are therefore usual references, i.e. no refinement references. Furthermore, also the disjunction of components into basic components and composite components has no effect on the system architecture.

The entire metamodel up to this point can be seen in Figure 5 where we again used dotted lines to represent reference refinements. The classes `Class`, `Reference` and `ReferenceType` are imported from NMeta and we omitted their details for brevity. Furthermore, we wrote in the attributes part of the classes also the constraints, both reference constraints and attribute constraints. Note in particular the attribute constraint of the `Interface` class that sets the `IsAbstract` attribute to True, meaning that instances of `Interface` as classes are abstract.

Because `AssemblyContext` is marked as an instance of `ComponentType`, we can generate a method that returns the component type of an assembly context. Furthermore, as the references of a component type are decomposed into its required interfaces, we statically know that each component instance will have an assembly context assigned for each required interface. Therefore, we can generate a method that takes a `RequiredInterface` and returns the corresponding `AssemblyContext` element connected on the reference generated for the required interface. The name of this method is constructed using the name of the reference that is used as a component for the references of a `ComponentType` as a class.

```
1  public interface IAssemblyContext : IModelElement, IReference {
2    IComponentType GetComponentType();
3    IAssemblyContext GetRequiredInterfacesValue(IRequiredInterface
         reference);
4  }
```

Listing 1: The generated interface for an `AssemblyContext`

The complete interface for an `AssemblyContext` is depicted in Listing 1. Any class generated for an instance of a component type implements this interface, resolving the component type object from a static URI. For more detail, we encourage the reader to have a look into the generated code for our usage example of DeepADL[3].

We believe, these generated methods make the generated API even simpler to use than with the workaround in traditional two-level modeling where developers of a model transformation would have to iterate manually through connector elements such as `AssemblyConnectors` in PCM to find the assembly context connected for a given required interface.

Assembly contexts are composed in a system architecture, i.e. a system architecture would instantiate components from a repository and connect them appropriately. Such a system architecture can be deployed to multiple resource environments. This deployment can be seen as an instantiation where the deployed system is an instance of the abstract system architecture. Thus, we model `SystemArchitecture` as a clabject, i.e. it inherits from `Class`. For each assembly context in a system architecture, developers must assign a resource

[3]http://github.com/georghinkel/DeepModelingDemo

container where the assembly context will be deployed to. Thus, the assembly contexts of a software architecture form its references as a class. Each such reference is typed with the resource container where the assembly context shall be deployed to.

This deployment view can be seen in Figure 6. In this diagram, the separation of the different modeling levels can be easily seen since the classes related to the deployment on the left hand of Figure 6 have no connection (except `InstanceOf`) to the classes representing the system architecture which are on the right hand.

A key observation here is that DeepADL contains classes spanning all levels involved in architecture description, i.e. repository, assembly and deployment. It is not restricted to the highest (repository) level. The purpose of classes on lower levels such as `AssemblyContext` for the system architecture level is mainly to give them an application-independent structure that is usable also for consumers of the model such as transformations or analyses.

Next, we describe the representation of composite components. Unlike basic components that are the smallest unit of implementation, a composite component bundles the functionality of multiple other components (possibly composite components as well), but does not contain any implementation on its own. In fact, composite components act like a Facade to their inner components. They are very similar to software architectures in that they contain some assembly contexts, i.e. instances of components, that are assembled together. On the other hand, composite components are components and thus belong to the repository level. However, the assembly of the inner components of a composite component is independent of the choice in which systems a composite component is used, if used at all. In particular, composite components require a reference to assembly contexts that are on a lower metamodelling level in Figure 4.

NMeta is agnostic of modeling levels and allows this. In particular, the metamodel fragment regarding composite components is depicted in Figure 7. Composite components may contain arbitrary many assembly contexts. These assembly contexts form the assemblies that the component uses to realize its functionality. A second reference `ExposedAssemblies` denotes the subset of assemblies that are exposed to outside world, i.e. the components that realize the interfaces that the component offers.

On the other hand, the component types of the assembly contexts used in the composite component may require interfaces. In order to have a valid model, all assembly contexts within the composite component must be connected to an instance of some class implementing the interface. This may be another assembly context within the composite component, but it may also be a delegation to the required interface of the composite component. While the first does not require any additional model elements, the latter requires to model delegation connectors explicitly. These delegation connectors must be instances of the interfaces which they delegate to. We model this as being instances of a common `Delegate` class. These delegates act like delegate type definitions in .NET that basically simply define a method signature. Likewise, delegators in DeepADL simply reference the interface which they delegate. Since they can be instantiated in delegation connectors, they are clabjects as well and thus inherit from `Class`.

This requires a new validity constraint as a delegation connector may only use a port with a type that is referenced from its delegate type.

This level-crossing makes it very hard to model composite components with level-adjuvant languages: Because conceptually, composite components may contain instances of composite components, any fixed amount of levels is not sufficient for deep-modelling composite components. Our structural decomposition approach is agnostic to levels and thus can ignore this problem. However, since one can only create an instance of a model element once it has been turned into a platform class, we can exclude that a composite component eventually contains an instance of itself, which is a validity constraint that is otherwise very hard to formulate.

6. EXAMPLE USAGE: A MEDIA STORE IN DEEPADL

The evaluate advantages of DeepADL over architectural languages using traditional two-level modeling, we consider a concrete example. In particular, we modeled a simple e-Commerce application called the MediaStore. This e-Commerce application lets users upload and download media files that are persisted in a database. Uploaded media files are processed with a watermark and saved to the database. This system has been used as a case study for PCM already in 2009 [6].

Since we do not have a suitable editor for models yet, we created small applications that use the generated model API to create models of the MediaStore system. These applications along with the generated code for all of the levels can be obtained online[4].

An overview of the MediaStore is depicted in Figure 8. This figure contains all three levels that have been discussed above. The components and the interfaces of the MediaStore form the repository. These repository components are then used to create the system architecture by composing them together. Finally, Figure 8 also shows the deployment that all components are deployed to a single application server except for the database component and the web browser.

In the Component Type Repository, the differences between PCM and DeepADL are small. This is reasonable since components are at a very high level. While PCM models provided interfaces explicitly through a `ProvidedRole` element, provided interfaces are referenced in DeepADL directly. However, if we take a look to the next level, the situation is different. The metamodel excerpt of PCM to represent the assembly contexts of a system architecture is depicted in Figure 9.

In PCM, assembly connectors are very generic. The fact that in a valid system architecture, each component must have an assembly connector for each required interface must be specified through an OCL constraint. Consequently, if a user forgets to add an appropriate assembly connector, he gets an error message saying that not all interfaces are connected unless a more appropriate error message is deducted from the OCL constraint or manually implemented. Conversely, one also needs to ensure that all assembly connectors of an assembly context are valid.

On the other hand, in the DeepADL version the classes that model editors are working with are much more specific to the MediaStore. Since they are in fact classes, we can visualize them in a class diagram. An excerpt of this class dia-

[4]http://github.com/georghinkel/DeepModelingDemo

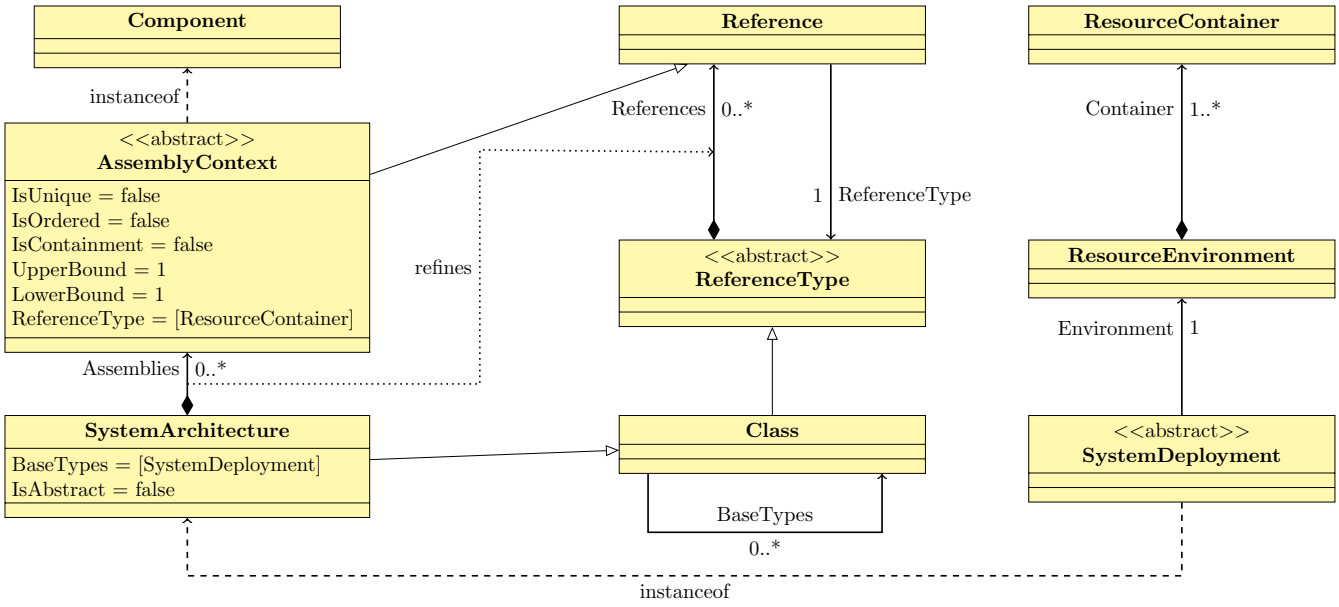

Figure 6: System deployment in DeepADL

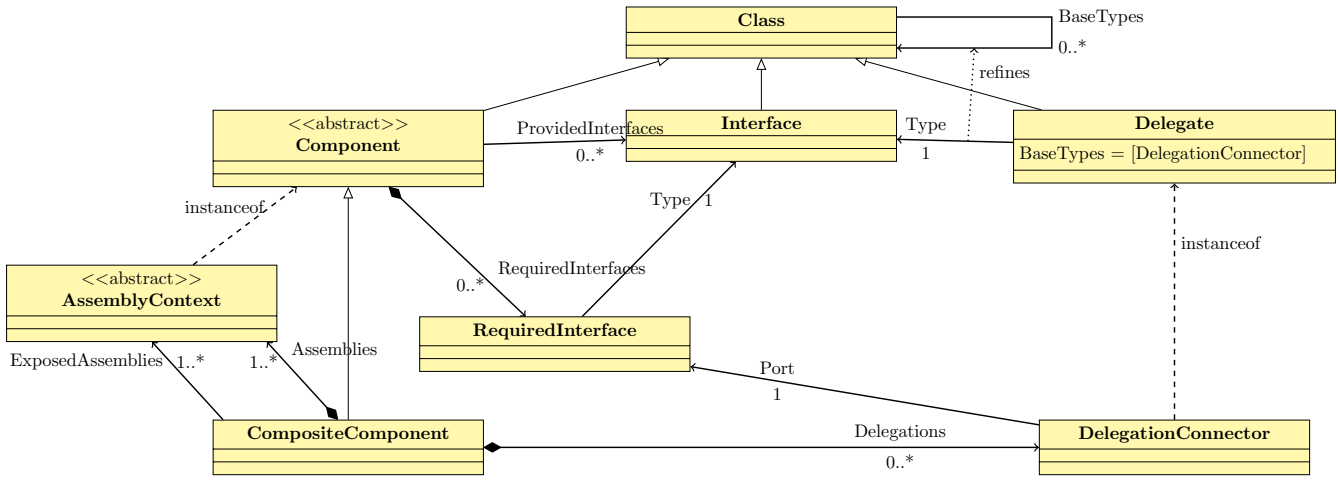

Figure 7: Composite components in DeepADL

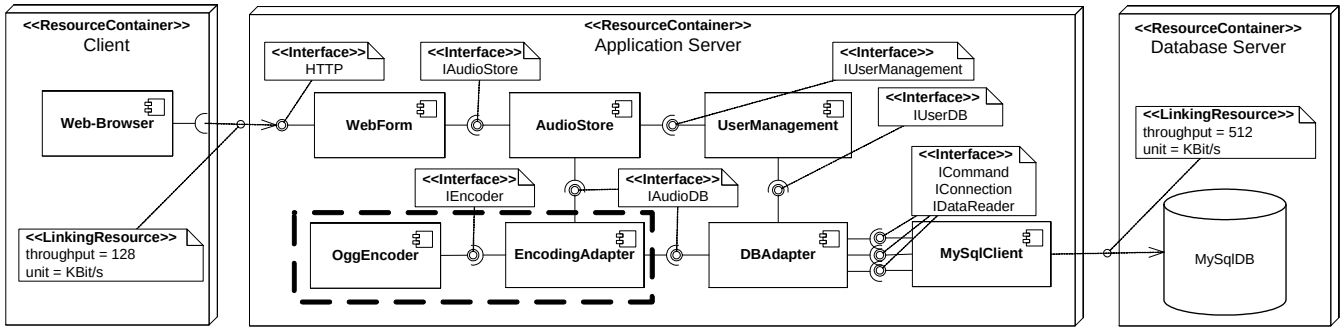

Figure 8: The software architecture of the MediaStore [6]

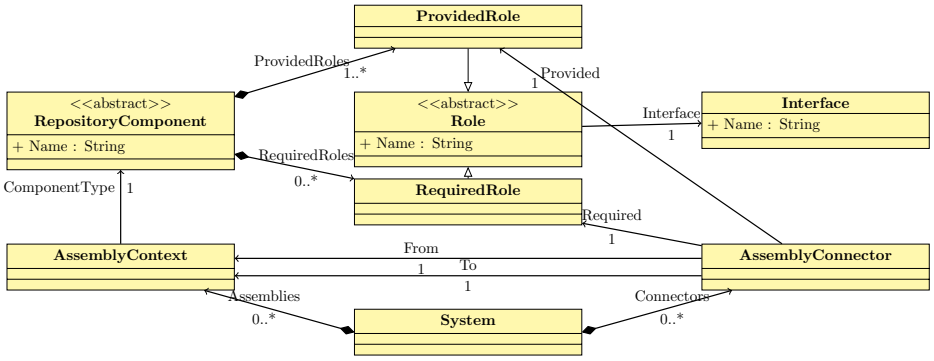

Figure 9: A simplified metamodel excerpt from PCM for system modeling

gram around the `AudioStore` component is depicted in Figure 10. Here, each required interface is turned into a reference because in fact a `RequiredInterface` model element *is* a reference. Each of these references has a multiplicity of 1, therefor implying a constraint that for example an `AudioStore` must have an `AudioDB` assigned. The error message that can be presented to the user if this contraint is violated, that the `AudioDB` of an `AudioStore` cannot be null, is much more specific and likely to be more helpful.

The NMF code generator for classes generates us a model representation code for the `AudioStore` component type, i.e. a generated API. We use this API to (currently programmatically) create instances of this component type. This generated API only allows us to set domain-specific properties like the referenced `IAudioDB` component but it does not show us class characteristics like references or attribute constraints. The reference of references has been refined whereas the attribute constraints reference has been constrained. In fact, the `AudioStore` class does not inherit from `Class` and explicitly implements its interface.

A similar statement is true for the deployment. Here again, the solution in two-level modeling such as applied in PCM is to introduce a generic concept of an allocation context. An excerpt of the PCM metamodel regarding the deployment can be found in Figure 11.

On the other hand, the deployment for the MediaStore in DeepADL is depicted in Figure 12. Because we created a SystemArchitecture called `MediaStoreSystem`, we can instantiate this clabject for the deployment of the MediaStore. Because an assembly context is a reference, we get a reference for every assembly context that is used inside the system.

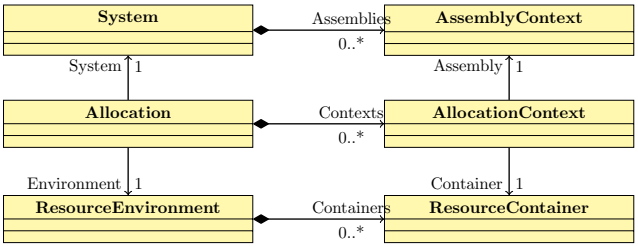

Figure 11: A simplified metamodel excerpt of PCM for deployment

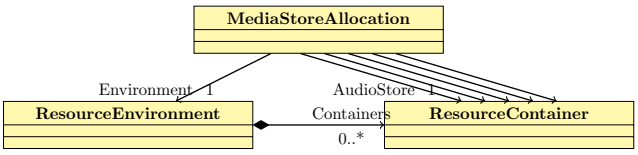

Figure 12: Deployment of the MediaStore in DeepADL

We also believe although we have not performed adequate empirical experiments that users find it more natural and intuitive to assign a resource container to a reference named *AudioStore* than creating an allocation connector that connects the AudioStore assembly with the resource container. This is even more true at the system architecture level where the user is asked to assign an instance of *IAudioDB* to the reference *AudioDB*. Here, the type system already checks

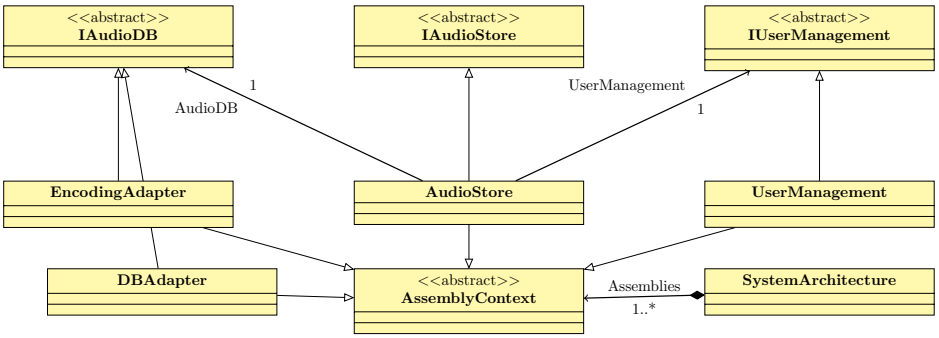

Figure 10: Excerpt of the implied DeepADL metamodel to specify the architecture of a MediaStore system based on a repository of component types

the validity so we can easily provide the user with tool support that only suggests valid options to connect, so only *DBAdapter* or *EncodingAdapter* instead of the all the components that appear in the repository.

Of course, all of this tool support can also be provided with two-level modeling techniques. However, to the best of our knowledge, there is no satisfying solution yet that analyzes the OCL constraints and uses this analysis to provide tool support up to such a level. The problem is that this information is widespread among a multitude of OCL constraints. The metamodel excerpt of Figure 11 needs one constraint to be valid, the excerpt from Figure 9 already four. In DeepADL, the same situation does not require a single OCL constraint.

7. RELATED WORK

The term of Multi-level modeling has got blurred in the recent past [11]. In the first workshop on Multi-Level-Modeling in 2014, the term has been replaced by Deep Modeling in order to also allow level-blind approaches such as ours, and to distinguish the field from statistics.

7.1 Refinements

The idea to use refinements for deep modeling is not new as in particular, Back and Von Wright have written a whole book on refinement calculus with a strong mathematical foundation based on lattices and set theory [12]. A usage in a model-driven context has been proposed by Varró and Patarisza in 2003 [13] or Pons [14]. However, these approaches have not tackled to represent instantiation chains using refinements.

7.2 Level-adjuvant languages

Level-adjuvant approaches typically use a level-agnostic meta-metamodel [15] describing the model structure. Many of these approaches are much more mature than ours and already provide rich tool support [16], [17]. However, we believe the case of composite components as in PCM is a case that inherently asks for support for level-crossing references as we presented in this paper, which is typically not supported by level-adjuvant languages.

On the other hand, Rossini et al. have created a comparison where they compare the level-adjuvant MetaDepth language with strict two-level modeling for cloud-based applications [18]. We believe that their solution using potencies is much simpler than a solution using structural decomposition could be, mainly because the potencies allow attributes clear when designing the first level to span multiple modeling levels more easily. Thus, there seems to be a trade-off decision between our approach and level-adjuvant languages when to apply which strategy.

7.3 Level-blind languages

In contrast to level-adjuvant languages, level-blind languages have no explicit notion of levels. Rather, they are helping constructs as the result of stratification. As our approach shows, this still allows to cross the boundaries of modeling levels. Atkinson et al. have presented some paradoxa for level-blind languages [11] for which they regarded these approaches to be logically inconsistent but because our approach requires an explicit mapping from a clabject to the platform type system, paradox situations like the quoted phenomena of being his own baby simply do not apply to our approach.

However, our approach is not the first one to be level-less. Henderson, Clark and Gonzalez-Perez have been working on an approach involving basically only objects and slots [19], [20]. However, this approach is not compatible to existing two-level metamodels such as Ecore and less convenient in terms of a generated API.

8. CONCLUSION AND OUTLOOK

In this paper, we have proposed an approach how deep modeling can be achieved with a slight and non-invasive extension to existing and well-accepted meta-metamodel such as Ecore. This brings us into the comfortable situation that we can apply deep modeling techniques such as non-transitive instantiation chains of arbitrary length with a self-describing and thus sound meta-metamodel and at the same time reuse all available tools to work with the models such as model transformation languages. At the same time, our approach circumvents paradox situations level-blind approaches to deep modeling have been blamed for in the past. We have applied our approach to the realistic scenario of creating a deep modeling version prototype of the popular Palladio Component Model where we could simplify the generated API for the model and reduce the number of constraints necessary. This case study also employs a case which crosses the boundaries of metamodeling levels, which is why we think that this case study will be hard to model in level-adjuvant languages.

In the future, we plan to create a generic model editor that allows users to create such models through a convenient interface.

References

[1] J. D. Lara, E. Guerra, and J. S. Cuadrado, "When and how to use multilevel modelling," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 12, 2014.

[2] P. Mohagheghi, W. Gilani, A. Stefanescu, and M. A. Fernandez, "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases," *Empirical Software Engineering*, vol. 18, no. 1, pp. 89–116, 2013.

[3] M. Staron, "Adopting model driven software development in industry–a case study at two companies," in *Model Driven Engineering Languages and Systems*, Springer, 2006, pp. 57–72.

[4] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, ACM, 2013, pp. 1–18.

[5] C. Atkinson, R. Gerbig, and C. Tunjic, "Towards multi-level aware model transformations," in *Theory and Practice of Model Transformations*, Springer Berlin Heidelberg, 2012, pp. 208–223.

[6] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, pp. 3–22, 2009.

[7] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bi-directional tree transformations: a linguistic approach to the view update problem," *SIGPLAN Not.*, vol. 40, no. 1, pp. 233–246, 2005.

[8] C. Atkinson, "Meta-modelling for distributed object environments," in *Enterprise Distributed Object Computing Workshop [1997]. EDOC'97. Proceedings. First International*, IEEE, 1997, pp. 90–101.

[9] J. J. Odell, "Power types," *Journal of Object-Oriented Programming*, vol. 7, no. 2, p. 8, 1994.

[10] B. Henderson-Sellers and C. Gonzalez-Perez, "Connecting powertypes and stereotypes.," *Journal of Object Technology*, vol. 4, no. 7, pp. 83–96, 2005.

[11] C. Atkinson, R. Gerbig, and T. Kühne, "Comparing multi-level modeling approaches," *MULTI 2014–Multi-Level Modelling Workshop Proceedings*, p. 53, 2014.

[12] R.-J. Back and J. Von Wright, *Refinement calculus: a systematic introduction*. springer Heidelberg, 1998.

[13] D. Varró and A. Pataricza, "Vpm: a visual, precise and multilevel metamodeling framework for describing mathematical domains and uml (the mathematics of metamodeling is metamodeling mathematics)," *Software and Systems Modeling*, vol. 2, no. 3, pp. 187–210, 2003.

[14] C. Pons, "Heuristics on the definition of uml refinement patterns," in *SOFSEM 2006: Theory and Practice of Computer Science*, Springer, 2006, pp. 461–470.

[15] C. Atkinson and T. Kühne, "Meta-level independent modelling," *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*, pp. 12–16, 2000.

[16] C. Atkinson and R. Gerbig, "Melanie: multi-level modeling and ontology engineering environment," in *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, ACM, 2012, p. 7.

[17] J. De Lara and E. Guerra, "Deep meta-modelling with metadepth," in *Objects, Models, Components, Patterns*, Springer, 2010, pp. 1–20.

[18] A. Rossini, J. Lara, E. Guerra, and N. Nikolov, "A Comparison of Two-Level and Multi-level Modelling for Cloud-Based Applications," in *Modelling Foundations and Applications: 11th European Conference, ECMFA 2015*, Springer, 2015, pp. 18–32.

[19] B. Henderson-Sellers, T. Clark, and C. Gonzalez-Perez, "On the search for a level-agnostic modelling language," in *Advanced Information Systems Engineering*, Springer, 2013, pp. 240–255.

[20] T. Clark, C. Gonzalez-Perez, and B. Henderson-Sellers, "A foundation for multi-level modelling," in *MULTI 2014–Multi-Level Modelling Workshop Proceedings*, 2014, p. 43.