# Refinements and Structural Decompositions in Generated Code

Georg Hinkel[1], Kiana Busch[2] and Robert Heinrich[2]
[1]*Software Engineering Division, FZI Research Center of Information Technologies, Karlsruhe, Germany*
[2]*Software Design & Quality Group, Karlsruhe Institute of Technology, Karlsruhe, Germany*
*hinkel@fzi.de, {kiana.rostami,robert.heinrich}@kit.edu*

Keywords: Metamodel, Refinements

Abstract: Todays systems are often represented by abstract domain models to cope with an increased complexity. To both ensure suitable analyses and validity checks, it is desirable to model the system in multiple levels of abstraction simultaneously. Doing so, it is often desirable to model that one association is a refinement of another to avoid duplication of concepts. Existing approaches that support refinements request metamodelers to use new modeling paradigms or have less efficient model representations than commonly-used technologies such as EMF with Ecore. In this paper, we propose a non-invasive extension to support refinements and structural decompositions in Ecore-like meta-metamodels, show how these extension can be supported by code generation and show that the fulfillment of refinements can be guaranteed by the underlying type system.

## 1 Introduction

In many engineering disciplines, abstract models of systems are created in order to reason on properties of the modeled system by analyzing the model. Many of such systems are nowadays supported by software that runs these analyses automatically based on in-memory representations of the models.

The structure and properties of these models are described again in a model, called the metamodel. Thus, the metamodel defines the level of abstraction followed in the system model. However, it is often a challenge to choose the most appropriate level of abstraction for such a metamodel. If the metamodel is too general, it may easily allow instance models that do not correspond with the real system. In such case, model validation rules may help to reduce this risk, but they also can only be specified using features contained in the metamodel. If features are specified in too specific subclasses, it gets hard to specify analyses because of case distinction.

To compensate for this problem, the UML (Object Management Group (OMG), 2015) has introduced concepts of refinements between associations in the form of subsetting, specialization and redefinition. This specification is also reused in the Complete Meta-Object Facility (CMOF) standard (Object Management Group (OMG), 2016). Though the semantics of these declarations is not clear from the standard, several works (Nieto et al., 2011; Costal et al., 2011; Hamann and Gogolla, 2013) have defined semantics of these definitions and implemented them in OCL constraints. However, the interaction of these subsetting, specialization and redefinition with other constraints such as multiplicity constraints have been a source of various problems (Maraee and Balaban, 2011; Maraee and Balaban, 2012), as the semantics turns out to be inconsistent.

In commonly used meta-metamodels such as Ecore, the most popular workaround is to create a feature in the most general concept and create derived features in more specific classes. An example for this is in Ecore itself, where `ETypedElements` simply have a type. More specific classes such as `EAttribute` or `EReference` inherit this reference, even though they could be more specific: The type of an attribute must be a data type or enumeration, while a reference always must be typed with a class. The metamodeler has to enforce this using a model validation constraint and this constraint has to be checked.

In an industrial context such as automated production systems, we see a very similar effect where sensors are generally equipped with a power supply, but some kinds of sensors only accept certain power supplies. Here, one would like to gain the expressiveness to specify the correct power supply type without having to use case distinctions in the analyses. Because a wrong power supply may have dramatic consequences in the physical sensor, we would like this constraint to be enforced as early as possible. Further, we also see the case that the exact type of the sensor is not known, for example because the sensor is supplied by a vendor.

Existing approaches to simplify the metamodel in this regard such as VPM (Varró and Pataricza, 2003), Deep Modeling tools (Atkinson and Gerbig, 2012; De Lara and Guerra, 2010) or CORE (Schöttle and Kienzle, 2015) require completely new modeling paradigms that mostly break existing tools. However, the availability of stable tools is one of the major factors for the lack of industry-adoption of model-driven engineering (MDE) (Staron, 2006; Mohagheghi et al., 2013). In addition, Meyerovich et al. (Meyerovich and Rabkin, 2013) have shown that most developers only change their primary language when either there is a hard technical project limitation or there is a significant amount of code that can be reused. Therefore, we suspect that many metamodelers would still like to use their usual meta-metamodel.

In this paper, we propose a formal foundation how refinements of associations can be implemented in a non-invasive way into an Ecore-like meta-metamodel. The proposed approach is able to guarantee that the refined associations are modeled correctly through guarantees of the target platform type system. This removes the potentially costly validation of an OCL constraint and more importantly, means that model analyses can completely rely on the fulfillment of these constraints. In the power supply example above, the modeler gets an immediate feedback in the form of an exception as soon as he tries to add a non-appropriate power supply to a sensor. We implemented our approach in the meta-metamodel NMETA that is part of the .NET Modeling Framework (Hinkel, 2016b) and discuss its advantages over alternative metamodel fragments in the domain of production automation.

The remainder of this paper is structured as follows: Section 2 briefly introduces our notion of refinements and structural decomposition. Section 3 gives a summary of the implementation of this concept in the meta-metamodel NMETA similar to Ecore. Section 4 explains how the concept can be applied to examples from the domain of production automation. Section 5 explains how code generators need to adjusted to support refinements and structural decomposition. Finally, Section 6 discusses related work before Section 7 concludes the paper.

## 2 Structural Decomposition and Refinements

In this section, we briefly introduce 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.

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. $\boldsymbol{A} \preceq \boldsymbol{B}$ if $\boldsymbol{B}$ is an ancestor of $\boldsymbol{A}$. We denote collections of type $\boldsymbol{A}$ with $\boldsymbol{A}*$ with concatenation operator ; similar to Kleene closures. Type membership is denoted with the usual $\in$ symbol.

In the formalization, a feature $f$ of a class $\boldsymbol{A}$ with type $\boldsymbol{B}$ is a persistent in-model lens (Hinkel and Burger, 2017) $f : \boldsymbol{A} \hookrightarrow \boldsymbol{B}$ consisting of a getter function $f \nearrow: \boldsymbol{A} \to \boldsymbol{B}$ and a setter function $f \searrow: \boldsymbol{A} \times \boldsymbol{B} \to \boldsymbol{A}$.

**Definition 1** (Structural decomposition)**.** *Let* $\boldsymbol{A}$ *and* $\boldsymbol{B}$ *be types. A list of features* $f_1, \dots, f_n : \boldsymbol{A} \hookrightarrow \boldsymbol{B}*$ *for types* $\boldsymbol{A}$ *and* $\boldsymbol{B}$ *and* $n \in \mathbb{N}$ *is a* structural decomposition *of a feature* $f : \boldsymbol{A} \hookrightarrow \boldsymbol{B}*$ *if we have that for each* $a \in \boldsymbol{A}$ *that*

$$f \nearrow (a) = f_1 \nearrow (a); f_2 \nearrow (a); \dots ; f_n \nearrow (a).$$

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

Since there is an embedding from $\boldsymbol{A} \to \boldsymbol{B}$ into $\boldsymbol{A} \to \boldsymbol{B}*$, we will also allow the features used for decomposition to be single-valued where we depict an element $\perp \in \boldsymbol{B}$ that corresponds to an empty sequence in $\boldsymbol{B}*$[1]. Similarly, 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 $\perp$.

**Definition 2** (Refinement)**.** *Let* $\boldsymbol{A}$*,* $\boldsymbol{B}$*,* $\bar{\boldsymbol{A}}$ *and* $\bar{\boldsymbol{B}}$ *be types with* $\bar{\boldsymbol{A}} \preceq \boldsymbol{A}$ *and* $\bar{\boldsymbol{B}} \preceq \boldsymbol{B}$*. Further, let* $f : \boldsymbol{A} \hookrightarrow \boldsymbol{B}$ *and* $g : \bar{\boldsymbol{A}} \hookrightarrow \bar{\boldsymbol{B}}$ *be features. Then, we say that* $g$ *is a refinement of* $f$ *if* $f \nearrow$ *and* $g \nearrow$ *are the same on* $\bar{\boldsymbol{A}}$*, i.e. the following equations holds for all* $a \in \bar{\boldsymbol{A}}, b \in \bar{\boldsymbol{B}}$*:*

$$g \nearrow (a) = f \nearrow (a)$$
$$g \searrow (a,b) = f \searrow (a,b).$$

*In particular, we know that for each element* $a$*, the reference* $f$ *will always refer to an element of* $\bar{\boldsymbol{B}}$*.*

[1]In implementations, $\perp$ is typically `null`.

An important special case here is the refinement by a constant reference $g \equiv b$ for some constant element $b \in I(\bar{\boldsymbol{B}})$[2]. Usually, constant features are not explicitly modeled as they do not contain any information specific to an instance, 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 Implementation in NMETA

We have implemented structural decomposition and refinements in the NMETA meta-metamodel, the meta-metamodel within the .NET Modeling Framework (Hinkel, 2016b). In the meta-metamodel, the support for refinements and structural decomposition simply adds a reference *Refines* of the `Reference` class to itself. The semantics of such an assignment is given by Definition 2. If multiple references refine a reference, that reference is structurally decomposed and the components are refined[3]. Therefore, the refined reference must be declared in an ancestor of the current class and the reference type of the refinement reference must be a descendent of the refined references type.

A reference may also refine a reference that is already refined in an ancestor class. In that case, the original reference is structurally decomposed by all references that refine the original reference. In other words, a decomposition is always scoped for a given class. Alternatively, a reference may also refine a reference that in turn refines another reference: Therefore, refinements can be stacked together.

Additionally, the metamodeler can add a constant reference into this structural decomposition through a dedicated model element called `ReferenceConstraint`. This means that the attribute is also refined by a constant model element or a collection thereof, in case the reference is typed with a collection. Only a single `ReferenceConstraint` is allowed per class and reference.

The class `ReferenceConstraint` is only necessary because NMETA has no support for derived features, yet, because it lacks a support for OCL. Using derived features such as in Ecore, one could use a derived reference instead that simply returns a constant value, therefore making the `ReferenceConstraint` class obsolete.[4]

For attributes, structural decomposition works the same but the types must match exactly because the inheritance hierarchy is not modeled. For this reason, a refinement alone is not practical for attributes, except for the case where attributes are refined with a constant attribute. Similar to references, this can be used to amputate features of a derived class, i.e. features that must not be set or must always have the same value if the object is of a given type.

For any composition of attributes or references, the multiplicity of the composition must be compatible with the multiplicity of the original feature. This means that the lower bound of the original feature must be smaller or equal to the sum of the lower bounds of components. Likewise, the upper bound of the original feature must be larger or equal to the sum of upper bounds of the components. Furthermore, we require that refinements of compositions are compositions. If a reference with an opposite is refined, we require that any refining reference has an opposite that refines the opposite of the original reference.

## 4 Examples in the Automated Production System Domain

This section provides further details on the automated production system example given in the introduction section. In particular, we investigate a range of modeling problems.

The domain of automated production systems involves software, as well as mechanic and electric parts. In this section, we use the example of sensor and power supply. In general, a sensor has a power supply. Several types of power supply exist, for example Alternating Current (AC) or Direct Current (DC). Additionally, there are several types of a sensor such as photoelectric, capacitive, or AC current sensors. Consider the example, that an AC current sensor must have an AC power supply, as illustrated in Figure 1.

This fact must be specified either through an OCL constraint or using derived features in an Ecore model. This is problematic because it is not clear whether or not other artifacts may depend on it. Some OCL constraints can be enforced automatically, but whether this is done is unclear from the perspective of a model analysis or transformation.

Using NMETA, we create an additional reference called *acPowerSupply* in the `ACCurrentSensor` class and set it to refine the original *powerSupply* reference. As a result, the generated implementation class for

[2]Here, the $\equiv$ symbol means that the getter function always returns the same element $b$.

[3]This definition is consistent because a single feature is a structural decomposition of itself.

[4]The reason for NMETA to not support derived features is that NMF has no support for OCL due to an incompatible code generation infrastructure.

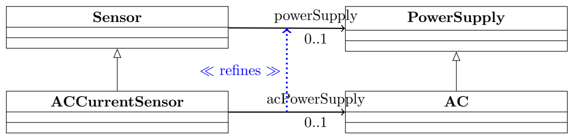

Figure 1: Modeling that an `ACCurrentSensor` requires an `AC` power supply in NMETA. The inserted refinement is printed in blue.

`ACCurrentSensor` will not inherit the more general power supply field, but includes a specific field to reference an `AC` element, from which the *powerSupply* reference is populated upon request. Because both references are single-valued, this just means that an `ACCurrentSensor` element returns its *acPowerSupply* when the power supply is requested. If an attempt is made to set the power supply, the generated code checks whether the provided power supply is an `AC` element and throws an exception otherwise, giving the modeler an immediate feedback.

Therefore, regular type system rules *guarantee* that a model where an `ACCurrentSensor` has a `DC` power supply cannot exist.

There are also sensors that do not need any power supply. For example, a surface acoustic wave sensor (Pohl, 2000) obtains its energy from piezoelectric and pyroelectric effects. We want to ensure, that the type system does not allow to model a case, in which a sensor with no power supply has an AC or DC power supply as illustrated in Figure 2.

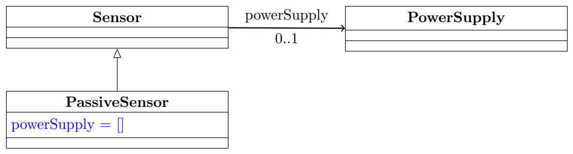

Figure 2: Modeling that a `PassiveSensor` must not have a power supply in NMETA. The inserted reference constraint is printed in blue.

Using refinements, this can be modeled by refining the the power supply reference with a constant. This works very similar to the refinement by other references but the power supply feature will return a constant instead of a different reference.

For references with higher cardinality, we face the problem that very general references can be decomposed in more specific subclasses. As an example, consider the power supply of a motor in a star or delta connection, as illustrated in Figure 3. The star-connected motors have a central point, where the similar ends of the wires are connected, whereas in the delta-connected motors the opposite ends of wires are connected. Thus, the delta connection results in a higher torque and a higher motor speed. However, some motors require both connections. For example, a three phase squirrel cage motor[5] has to be started in a star connection. After the normal speed is reached, it has to be switched from a star to a delta connection.

To model this situation, using NMETA we can simply mark multiple references to refine the same reference. We call this a structural decomposition. In this case, when a `3PhaseSquirrelCageMotor` is asked for its connections, it assembles this list on the fly from its star connection and its delta connection. When an element is to be added to the *connected* reference, the generated code for a `3PhaseSquirrelCageMotor` checks whether that connection is either a `StarConnection` or a `DeltaConnection` and adds it to the respective reference. If multiple references apply, then the element is added to the first reference that is not already full with respect to its cardinality.

## 5 Code Generation

Similar to EMF, NMF provides a code generator for metamodels (Hinkel, 2016b). Because the metametamodel allows multiple inheritance, the code generator generates both an interface and a default implementation class for each class in the metamodel, again similar to the EMF code generator. To keep the generated code small, the code generator reuses default implementation classes for subclasses as much as possible.

For any attribute or reference (feature in the remainder), a property and a change event is generated. If the featuire is not refined, this property is backed by a field. In case a feature is refined, a private getter and setter implementation[6] is generated instead that composes or decomposes the property on the fly.

Therefore, refinements impact the inheritance hierarchy of the implementation base class. In case a feature is refined, the code generator may no longer reuse any implementation class that contains a backing field for this feature.

For example, consider the classes `Sensor` and `ACCurrentSensor`. The code generator generates an interface and a default implementation class for each of these classes in the metamodel. Because there is an inheritance relation between the metaclasses, the generated interface for

[5]`http://www.pcbheaven.com/userpages/check_the_windings_of_a_3phase_ac_motor/`, accessed 10 Jul 2017

[6].NET allows classes to privately implement an interface, which means that the implementation is not visible from the class API, but only through this interface.

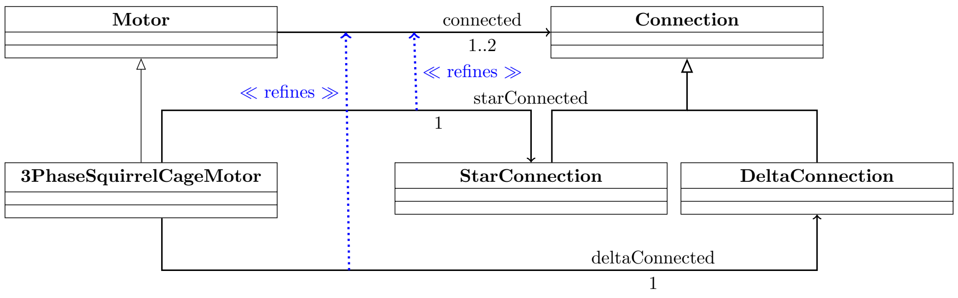

Figure 3: Star and delta connections in motors modeled in NMETA. The inserted structural decomposition is printed in blue.

`ACCurrentSensor` does inherit from the generated interface for `Sensor`. However, the generated implementation type `ACCurrentSensor` does not inherit `Sensor`, in order to avoid inheriting the *powerSupply* reference backing field.

On the other hand, the code generator may still reuse an implementation class for base classes of `Sensor` to avoid replicating too many features.

We may encounter diamond problems. If, for the sake of the example, a class inherited from both `ACCurrentSensor` and `PassiveSensor`, the code generator must not reuse the implementation class of either `ACCurrentSensor` or `PassiveSensor`, because the decomposition of the feature *powerSupply* is different to the one in `ACCurrentSensor` or `PassiveSensor`, where only one feature refines *powerSupply*.

As soon as an appropriate base class is found, we may simply copy the generated code for the features that cannot be inherited and copy them into the generated type, as long as they are not refined.

To find such a base class, we propose the abstract algorithm depicted in Algorithm 1. It is essentially based on a reversed topological order of strongly connected components in a dedicated graph that is induced by inheritance relations and refinements. The result of Algorithm 1 for a class $c$, if not $\perp$, is a class $c_b$ that

- is a base class of the class to be generated ($c \preceq c_b$).
- only contains properties that have not been refined by classes $\tilde{c}$ with $c \preceq \tilde{c} \prec c_b$.
- that does not contain a decomposition that is no longer valid. Here, an invalid decomposition refers to a decomposition of a feature $f$ into $f_1;\ldots;f_n$ in the scope of a class $\tilde{c}$ with $c \prec \tilde{c}$, but $f$ is decomposed in $c$ by a larger list of features.

In Algorithm 1, the function ALLFEATURES simply computes the set of all attributes and references available in a given class, including inherited and transitively inherited. The function REFINEMENTS returns those attributes and references that are refined by attributes or references of the given class. More interesting is the function EDGE that defines the edges in the graph the topological order is created for. This graph shall contain edge from a class $c_s$ to a class $c_t$, if the generated code for class $c_s$ obsoletes the generated code for $c_t$. This may either be because $c_s \preceq c_t$ or because $c_s$ refines a property of $c_t$. The latter case is not problematic for the case that $c_t \preceq c_s$, because in that case, the generated code for $c_t$ is aware of this refinement.

The reversed topological order guarantees us that there is no incoming edge from ancestor classes not yet considered for the given class. It can be easily implemented by reversing the output of Tarjan's algorithm (Tarjan, 1972). In Algorithm 1, we assume the latter to return a list of strongly connected components, each represented as set of classes.

The graph may contain cycles. Because inheritance is acyclic and we only allow features to refine features of base classes, such a cycle must come from a set of classes that refine features of a common base class, i.e. we are facing a diamond-shaped inheritance. Because the generated code for the bottom of the diamond must respect all refinements made in any of its base classes, no generated code for a class contained in a cycle must be reused. However, there still may be a common ancestor class whose features have not been refined, such as any base class of `Sensor`.

Applied to the class `ACCurrentSensor`, the reverse topological sort returns the strongly connected components $\{$`ACCurrentSensor`$\}, \{$`Sensor`$\}$. `Sensor` is not chosen, because its property *powerSupply* is refined.

Because the class `ACCurrentSensor` does not inherit an implementation of `ISensor`, it implements this interface directly by duplicating the implementation of non-refined attributes and references. This code duplication is not problematic since the code is generated and therefore not manually maintained. The metamodel as the source from the code generation does not have this duplication. The *powerSupply*-reference is implemented in private where the getter

**Algorithm 1** Find implementation base class

```
function $\textsc{AllFeatures}(c)$ return $\bigcup_{c \preceq c_b} c_b.Attributes \cup \bigcup_{c \preceq c_b} c_b.References$
function $\textsc{Refinements}(c)$ return $\{g | f \in c.Attributes \cup c.References, f \xrightarrow{\ll refines \gg} g\}$
function $\textsc{Edge}(c_s, c_t)$ return $c_s \preceq c_t \vee (\textsc{Refinements}(c_s) \cap \textsc{AllFeatures}(c_t) \neq \emptyset \wedge c_t \not\preceq c_s)$
function $\textsc{FindBaseClass}(c)$
    $shadows \leftarrow \emptyset$
    $ancestors \leftarrow \textsc{TransitiveHull}(c, cl \mapsto cl.BaseTypes)$
    for all $layer$ in $\textsc{ReverseTopologicalOrder}(ancestors, \textsc{Edge})$ do
        if $|layer| = 1 \wedge layer \neq \{c\} \wedge shadows \cap \textsc{AllFeatures}(layer[0]) = \emptyset$ then return $layer[0]$
        for all $l$ in $layer$ do
            $shadows \leftarrow shadows \cup \textsc{Refinements}(l)$
    return $\perp$
```

simply returns the *acPowerSupply* reference and the setter tries to cast the value appropriately and sets the `acPowerSupply` reference, if applicable and throws an exception otherwise.

For a structural decomposition such as the *powerSupply* reference of `3PhaseSquirrelCageMotor`, a dedicated collection class is generated. When browsed, this collection class iterates all the components in sequence. In the example, the iterator first returns the star connection (if not `null`) and then the delta connection (if not `null`). In case one of the component features is multi-valued itself, all items of this component are iterated. Conversely if a new element is added to the composite feature such as *powerSupply*, the generated code tries to find a component reference to which the element can be added, based on the element type and the cardinality of the component reference.

Artifacts that modify model elements such as editors would rather operate on the real type of the model elements and therefore only see the public properties, which are exactly the non-decomposed and non-refined properties.

By default, the generated XMI code for a serialized model will not contain any information on decomposed features since they can be reconstructed by its components. If information about refinements is cut off (e.g. by exporting the metamodel to Ecore), the serialization simply needs to be configured to serialize also refined features and then tools not aware of structural decomposition are able to read the model just like any other model. Conversely, when reading the model, the refined features are just ignored in the deserialization, such that models created by tools not enabled for structural decomposition can be loaded – the only problem here is that these tools may unnecessarily demand the modeler to specify features that are otherwise refined.

Therefore, existing tools not aware of structural decomposition and refinements can be reused without changes, though they may not offer the best convenience.

## 6 Related Work

We see related work in the areas of refinements, deep modeling languages (in particular level-adjuvant ones) and aspect-oriented modeling and discuss them in the following sections.

### 6.1 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 (Back and Von Wright, 1998). A usage in a model-driven context has been proposed by Varró and Patarisza in 2003 (Varró and Pataricza, 2003) or by Pons (Pons, 2006). In contrast to our approach, they break with existing modeling paradigms. Furthermore, they do not seem to enforce the refinements through the type system.

The specifications of UML and CMOF also know refinements, as redefinitions and subsets. However, the actual semantics of these constructs is not detailed in the specification (Object Management Group (OMG), 2016).

The UML defines three methods to refine associations: redefinition, specialization and subsetting, though as mentioned, the semantics and especially their interplay are not clearly defined. In particular, these definitions have some correctness problems in connection with other constraints such as multiplicity constraints as shown by Maraee and Balaban (Maraee and Balaban, 2011; Maraee and Balaban, 2012). To some degree, our approach tackles the necessity of the three of those: Our implementation of refinements matches redefinition quite closely, but we show that through the interplay of refinements of the same features, we can support many more modeling scenarios. We also consider the interconnection of refinements with multiple inheritance.

There have been a couple of works to define the semantics of UML association refinements through OCL constraints (Nieto et al., 2011; Costal et al., 2011; Hamann and Gogolla, 2013). Closest to our approach, Nieto et al. (Nieto et al., 2011) propose a semantics for association redefinition and use a similar notation. However, they also implement refinements through a constraint of the more general reference and therefore do not inherit type-system guarantees. Furthermore, their approach is limited in that only a single reference may refine another reference.

Further, tools that use OCL to check the validity of models are usually dynamic in the sense that they represent models and metamodels in memory. Our approach uses a generative approach where code is generated from a metamodel to represent models in memory. A generative approach usually has a faster model API at the cost of higher maintenance efforts.

We are not aware of any solutions that considered UML redefinition in combination with diamond-shaped inheritance and how these situations can be resolved.

### 6.2 Level-adjuvant deep modeling languages

The possibility to refine references also has been implemented in level-adjuvant (Atkinson et al., 2014) deep modeling approaches that allow to refine references through an instantiation. Level-adjuvant approaches typically use a level-agnostic meta-metamodel (Atkinson and Kühne, 2000) describing the model structure. Many of these approaches are much more mature than ours and already provide rich tool support (Atkinson and Gerbig, 2012; De Lara and Guerra, 2010). However, the introduced potency-concept of these approaches is a breaking new concept that disallows the usage of existing tools and makes evolution scenarios hard.

### 6.3 Aspect-oriented modeling

Refinements are only one possibility to simplify the modeling of recurring patterns. Another possibility is to model the pattern once and very explicit, including possible constraints that have to be implemented, and weave this pattern implementation into a concrete use case using aspect-oriented modeling techniques. An example is CORE[7]. However, once applied, concerns have a fixed level of abstraction, while refinements allow to model multiple levels of abstraction concurrently. In our scenario, this is important in the case of bought components where their inner structure is not known.

## 7 Conclusion and Outlook

In this paper, we have proposed a formal definition of refinements and structural decomposition, how they can be implemented in a meta-metamodel and how a code generator can be designed to ensure them through type system guarantees. This can make many validation constraints obsolete as a demonstration of these concepts in the domain of industrial production automation shows.

Our notion of refinements matches redefinition of properties as defined in UML. However, UML redefinitions do not consider the case that multiple properties redefine the same property and currently, it is unclear what the semantics should be in that case and even whether this should be allowed at all. Here, our approach proposes a semantics by extending the semantics of the refinement information to structural decomposition. As our implementation shows, these semantics can be enforced by underlying type-system guarantees in case the metamodel is generated to code.

Because refinements and structural decomposition can be stacked, these concepts make it viable to model a system in a fine granularity in order to ensure correctness, while still being able to analyze the model at a high level of abstraction. We envision that metamodels may contain specific metaclasses down to a low level of abstraction, basically down to a level of manufacturers and makes of a certain type of component. While this allows very detailed correctness checks, it also bloats the metamodel and demands for concepts to specify the make of a component type in a different model. Therefore, we want to investigate how these problems can be solved using deep modeling ideas. Early experiments in this direction (Hinkel, 2016a) showed promising results.

## Acknowledgements

This work was supported by the DFG (German Research Foundation) under the Priority Program SPP 1593: Design For Future – Managed Software

[7]Concern-Oriented REuse (Schöttle and Kienzle, 2015)

Evolution and by the MWK (Ministry of Science, Research and the Arts Baden-Württemberg) in the funding line Research Seed Capital (RiSC).

## REFERENCES

Atkinson, C. and Gerbig, R. (2012). Melanie: multi-level modeling and ontology engineering environment. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, page 7. ACM.

Atkinson, C., Gerbig, R., and Kühne, T. (2014). Comparing Multi-Level Modeling Approaches. *MULTI 2014–Multi-Level Modelling Workshop Proceedings*, page 53.

Atkinson, C. and Kühne, T. (2000). Meta-level independent modelling. *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*, pages 12–16.

Back, R.-J. and Von Wright, J. (1998). *Refinement calculus: a systematic introduction*. springer Heidelberg.

Costal, D., Gómez, C., and Guizzardi, G. (2011). *Formal Semantics and Ontological Analysis for Understanding Subsetting, Specialization and Redefinition of Associations in UML*, pages 189–203. Springer Berlin Heidelberg, Berlin, Heidelberg.

De Lara, J. and Guerra, E. (2010). Deep meta-modelling with metadepth. In *Objects, Models, Components, Patterns*, pages 1–20. Springer.

Hamann, L. and Gogolla, M. (2013). *Endogenous Metamodeling Semantics for Structural UML 2 Concepts*, pages 488–504. Springer Berlin Heidelberg, Berlin, Heidelberg.

Hinkel, G. (2016a). Deep Modeling through Structural Decomposition. Technical report, Karlsruhe Institute of Technology, Karlsruhe.

Hinkel, G. (2016b). NMF: A Modeling Framework for the .NET Platform. Technical report, Karlsruhe Institute of Technology, Karlsruhe.

Hinkel, G. and Burger, E. (2017). Change Propagation and Bidirectionality in Internal Transformation DSLs. *Software & Systems Modeling*.

Maraee, A. and Balaban, M. (2011). On the Interaction of Inter-relationship Constraints. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa, pages 3:1–3:8, New York, NY, USA. ACM.

Maraee, A. and Balaban, M. (2012). *Inter-association Constraints in UML2: Comparative Analysis, Usage Recommendations, and Modeling Guidelines*, pages 302–318. Springer Berlin Heidelberg, Berlin, Heidelberg.

Meyerovich, L. A. and Rabkin, A. S. (2013). Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 1–18. ACM.

Mohagheghi, P., Gilani, W., Stefanescu, A., and Fernandez, M. A. (2013). An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical Software Engineering*, 18(1):89–116.

Nieto, P., Costal, D., and Gómez, C. (2011). Enhancing the semantics of UML association redefinition. *Data & Knowledge Engineering*, 70(2):182 – 207.

Object Management Group (OMG) (2015). Unified Modeling Language (UML) – Version 2.5 (formal/2015-03-01).

Object Management Group (OMG) (2016). MOF 2.5.1 Core Specification (formal/2016-11-01).

Pohl, A. (2000). A review of wireless SAW sensors. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, 47(2):317–332.

Pons, C. (2006). Heuristics on the definition of UML refinement patterns. In *SOFSEM 2006: Theory and Practice of Computer Science*, pages 461–470. Springer.

Schöttle, M. and Kienzle, J. (2015). Concern-Oriented Interfaces for Model-Based Reuse of APIs. In *Proceedings of the 18th International Conference on Model-Driven Engineering Languages and Systems - MODELS 2015*, pages 286–291. ACM.

Staron, M. (2006). Adopting model driven software development in industry–a case study at two companies. In *Model Driven Engineering Languages and Systems*, pages 57–72. Springer.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160.

Varró, D. and Pataricza, A. (2003). 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*, 2(3):187–210.