CMOF-Model Semantics and Language Mapping for MOF 2.0 Implementation

Markus Scheidgen
Institute of Computer Science, Humboldt-Universität zu Berlin
scheidge@informatik.hu-berlin.de

Abstract

Meta-modelling programming frameworks enable engineers to deal with models, defined through object-oriented meta-models, in the environment of programming languages. Existing frameworks use redefinition relationships between meta-model classes to encourage reusable meta-model design. In contrast to existing platforms the upcoming MOF 2.0 OMG recommendation proposes the meta-modelling language CMOF that also includes the possibility to define redefinition, and sub-setting constraints between the properties of meta-model classes.

In this paper we extend existing implementation strategies and language mappings to realize these new features in a MOF 2.0 implementation. We proposes a Java language mapping for the CMOF-model, based on method overwriting with changing return types and generic collection types that allow reasonable static type safety. Furthermore, we describe the semantics that are needed to implement functionality for adding and removing property values that automatically yields sub-setting constraints.

1. Introduction

Modern software systems are complex. To deal with this complexity engineers use abstractions. Abstractions are descriptions that leave out unnecessary detail and can be expressed in models. In software engineering object-oriented modelling is used for all aspects of a system: structure, behaviour, deployment etc.; modelling is used at different levels of detail. For instance, engineers distinguish between platform independent and platform dependant models. Engineering based on models has lead to Model Driven Development (MDD) with OMG's Model Driven Architecture (MDA) [24] as an widely used and excepted model centric development methodology.

The models used in such processes are written in distinct languages. Those languages define specific syntax and semantics for the concepts of a modelling language. Such modelling language definitions have to be precise in order to allow unambiguous understanding and they allow computer aided modelling with model editors or model transformations. Meta-modelling platforms [22] achieve this required precision by defining languages themselves through models. Concluding, those language models —meta-models—are also defined by models. This technique is usually used in for layers: data (M0), models (M1), meta-models (M2), and meta-meta-models(M3). To terminate this chain of models, the M3 layer is usually the top layer, modelled with its own concepts.

Such meta-modelling platforms are realized in numerous existing meta-modelling frameworks. Each modelling framework comes with a concrete M3 model that defines all meta-modelling concepts available in a concrete framework. Each framework defines functionality, APIs, and semantics to create and modify model elements. The Meta Object Facility (MOF) [25, 5] , for example, is best known as the modelling platform used to defined the Unified Modelling Language (UML) [6, 8]. MOF is a platform independent modelling framework, hence it is described with UML and abstract IDL interfaces. In contrast to MOF, the Eclipse Modelling Framework (EMF) [11] was develop as a framework that is more centred on programming with models. Therefore, it is tightly connected to an implementation language, which is in this case Java. [1]

In order to facilitate models in applications, like tools for computer aided model driven development, models defined in a distinct framework must be accessible in a programming language. Where frameworks like EMF already come with a programming language interface, do others, like MOF, need a language mapping that maps the abstract modelling concepts to a concrete programming platform. An IDL mapping [25, 15] and a Java mapping (JMI [21]) exist for MOF. These language mappings serve two purposes: first, they define an application programming interface for creating, modifying, or storing models, and second they ex-

[1] There is a different kind of frameworks, like the eXecutable modelling framework (XMF) [12], which feature a M3-model that is expressive enough to completely self-describe the semantics of the framework. The advantage is that those framework are completely closed systems that do not depend on an external programming environment. But they do not, at least for now, scale very well.

press model elements as objects in the programming environment, i.e. map M3-model concepts to the concepts of the chosen programming language.

With the next major release of UML also comes a new major release of MOF. The corresponding MOF 2.0 recommendations define the M3 model CMOF, which is used by the UML 2.0 recommendations to define the new UML with CMOF itself. In contrast to older MOF versions, and comparable frameworks like EMF, it introduces new relations to express specialisations between model elements. First, this is the redefinition of properties, like attribute or association ends, which allows to refine a property when it's owning classifier is specialized. Second, CMOF allows to define subset relationships between properties, which constrains the values of one property to form a subset of the value of another.

With this features, CMOF enhances modularity and the possible use of abstractions within meta-models and thus enforces the reuse of concepts. With this, in addition to CMOF's packet merges, UML 2.0 could be defined in its highly modular fashion, where each UML superstructure concept is a combination of abstractions provided by UML's infrastructure. These techniques, important to UML 2, are also important for the modular definitions of other languages, especially when they are based on UML, i.e. are UML-profiles.

Besides improvement in meta-modelling as in the CMOF-model, programming languages have advanced too. Existing Java language mappings of meta-modelling frameworks do not utilize important features of newer Java versions. Interesting here are covariant return types and generics. Covariant return types means that Java allows the return type of an overwriting method to be a specialisation of the overwritten method's type. Generics are templates of classes or interfaces; concrete realisation of such a template, replace placeholder types with concrete types. That way collections of elements can be written for a concrete element type with out knowing that type in advance. Those features should be used to create modelling frameworks that are safer to program (less casts and more static type checking) by using generic collection types for sets of property values, and covariant return types for the redefinitions of model features.

In this paper we solve the following two problems. First, existing Java language mappings do not include newer CMOF features, nor do they utilize generics or covariant return types. Second, in order to implement CMOF properly we need semantics for property sub-setting that can be implemented in a modelling framework that automatically ensures sub-setting constraints.

To solve these problems, we developed a CMOF based modelling framework, called *A MOF 2.0 for Java* [1]. It includes a Java language mapping. This mapping allows more static type checking at runtime than JMI or EMF do by using new Java features, and the mapping maps CMOF's new property redefinitions to Java's covariant return types. The framework includes a mechanism that uniformly handles associations and sub-setting constraints as relations between properties and thus can automatically ensure the implied subset constraints.

The following section 2 introduces existing implementation strategies for meta-modelling repositories and compares them to the work presented in this paper. Furthermore, it introduces other work that addresses the subject of CMOF language mappings and semantics. Section 3 describes a CMOF Java language mapping that uses covariant return types and generic collection types for a more convenient programming and enhanced type safety. Section 4 explains the details of property subsets and introduces semantics that allow uniform handling of associations and the automatic update of subset properties. The concluding section 5 visualizes the possible impact of this work on meta-modelling frameworks and suggest further work.

2. Related Work

Multiple meta-modelling frameworks and their implementations exist. We start with a brief introduction into two popular frameworks: EMF and MOF. After a short discussion of the ideas and solutions used in these frameworks and what of them could be reused in our implementation, we introduce other research work that is important to realize aspects that are not covered from existing frameworks.

The Eclipse Modeling Framework (EMF) [11] and frameworks based on the MOF 1.x recommendations [25] are widely used modelling frameworks. Gerber and Raymond have compared MOF and EMF in [16] to bridge the gap between the two communities that have assembled around MOF and EMF. The conclusions from Gerber and Raymond are that both frameworks are conceptually very similar, hence they are both frameworks for object-oriented meta-modelling; they model concepts with classes, typed attributes, allow multiple inheritance, modularisation with packages and package nesting. But where the EMF puts more emphasis on simplistic programming with meta-data and thus comes with a smaller meta-modelling language, MOF has richer meta-modelling concepts and puts emphasis on more expressive modelling.

The Java Metadata Interface (JMI) [21] presents a Java language mapping for the MOF 1.x recommendation, and thus gives Java users a meta-modelling framework based on the MOF, which does include a Java language mapping itself. JMI is implemented, among others, by SUN's Metadata Repository [3] and the ModFact project [2].

The three mentioned implementations of modelling frameworks EMF, MDR, and the ModFact repository offer a

common feature set and have all similar design. They offer a build in repository for meta-models, support code generation for user defined repositories based on meta-models. Models and meta-models can be loaded into repositories, using XMI [28] based import and export facilities. Model elements can be accessed using either generated interfaces or a meta-model independent reflection mechanism. All three implementations store models in an uniform and meta-model independent instance model, which is wrapped by a layer of delegators that presents the reflection functionality, and that is itself wrapped by generated typed interfaces (that extend the general reflection interfaces) and that allow access to the model via classes, properties and methods that have names and types based on a corresponding meta-model.

The basics of those frameworks are well understood and proven. Therefore, we use the same architecture in our framework, as well does our language mapping follow the general rules for a MOF based repository that are defined in JMI. However, the existing frameworks and their corresponding meta-modelling languages do not support the new MOF 2.0 features: refinement of classifier features and sub-settings between properties. Basically do existing platforms cover the smaller brother of CMOF called EMOF (essential MOF), which is tailored for simpler and easier to realize modelling. Another fact is that none of the existing platforms uses generic collection types for value collections, what usually leads to statically unsafe programming practices, including an annoying number of type casts and extensive use of runtime type interspection with the *instanceof* Java operator.

For the CMOF model, the MOF 2.0 IDL [15], issued by the OMG, presents a MOF language mapping to OMG's platform independent Interface Definition Language (IDL) [4]. Unfortunately this work is heavily constrained by limitations imposed through the very strict static type system of IDL. For instance, the type constraints implied by the redefinition of properties cannot be reflected in the target language, because IDL does not support operation overwriting with changing return type (covariant return types). This work presents value collection with concrete types. Since IDL does not have a build in mechanism for generic types, those collections are generated for each type defined in a meta-model. This idea can be improved to work with generic collection types in Java, and can be enhanced to exploit the possibility of covariant return types offered by the Java programming language.

Ameluxen et al. [10] developed templates of Java code that realize the semantics of sub-setting in association ends and attributes or more general in properties. Here the update of a property value also triggers updates to subset properties in order to keep values of properties subsets of each other. This can be generalized: sub-setting is only a special kind of constraint for property values, another constraint kind is imposed by associations.

The implementation of associations is an often discussed aspect in modelling. Associations can be implemented as objects in their own right, which realize a bi-directional relationship using references from the association object (link) two the linked objects. This approach is chosen in MOF/JMI and was favoured by Rumbaugh et al. [27, 16]. The approach, championed by Graham et al. and used in EMF [18, 16], realizes associations ends as references that are constrained to be opposites of each other. We use the second approach, because it is easier to unify with property sub-setting, which is just another way of constraining properties.

3. A Language Mapping

A language mapping is needed to use MOF based models in a programming language; it defines how model elements are represented by objects in the programming language, how such elements can be created, modified, or deleted using a programming language. A language mapping maps CMOF concepts to the concepts of a programming language. In case of the object-oriented CMOF-model and the object-oriented programming language Java, this is often straightforward: model elements are mapped to Java objects, their classifiers to interfaces and classes, CMOF properties to Java properties[2], and so on.

We choose to realize a CMOF mapping for the Java programming language, because Java offers a flexible type system. Compared with IDL (the recommended language to describe an interface to MOF), Java provides two important features that IDL does not: (1) *Covariant* return types that were introduced with Java Platform 2 Version 1.4 and (2) *generics*, introduced with Java Platform 2 Version 5. Those features will prove necessary for a convenient modelling API. The importance of covariant return types and generics will be discussed later in this section. For more information on static type checking and static semantics in object-oriented languages refer to [23, 17].

The previous MOF recommendation MOF 1.x [25] and its Java mapping defined in the *Java Metadata Interface(JMI)* [21], already constituted the common practices for a MOF to Java mapping. We propose a language mapping that follows these practices: Every model element is represented through a proxy object that implements the interface that corresponds to the according meta-class. To call operations on a model element, or to access its attributes, or linked objects, according Java methods have to be invoked. For more information about MOF semantics and language mappings refer to the old *MOF 1.4* [25] and *JMI* [21] or the

[2] Java properties are member variables, accessible through a pair of get- and set-methods

new *MOF2.0 Facility and Object lifecycle* [9] and *MOF2.0 IDL* [15] recommendations.

We extend this mapping to solve two problems: (1) JMI does not have support for CMOF's feature redefinitions, (2) JMI does not incorporate the possibility of generic types, hence programming with JMI often requires class-cast and reflection on runtime types. Common functionality, like model navigation, property updates, object creation, etc. has to be programmable in short and safe idioms. Following that rationale means that long chains of calls and numerous cast should not be necessary in the usual case.

The rest of this section will handle the detailed mapping of features; features are attributes, association ends, and operations. The next three subsections will cover (1) redefinitions of features, (2) merging of classes (a special application of feature redefinition), and (3) features with arbitrary multiplicity.

3.1. Mapping redefinitions

The UML 2 Infrastructure library (which includes abstract, basic definitions for the MOF model) allows an element to redefine other elements in the context of an generalisation relation. The abstract concept *redefinition* has different semantics and syntactic implications in its concrete realisations.

For properties it must hold that a redefining property is as least as restrictive as the redefined, or, in other words, the redefining property has to *be consistent with* the redefined property. In detail, the multiplicity of the redefining property must be included in the multiplicity of the redefined property, the redefining property must be derived when the redefined is, etc. The constraint that the type of a redefining property must be covariant (it *must conform*) to the type of the redefined property, is important for the language mapping. A property $p'{:}B$ of type B can redefine $p{:}A$ of type A, if B is a direct or indirect subtype of A, denoted as $A \lhd\!\!- B$. This is a *covariant type change*.

But for most implementation languages more restrictive rules apply. Because many of those languages try to assure static type safety, the type system has to be more restrictive. Some of these languages are Java, C++, and IDL. Most of those languages have the following, or even more restrictive redefinition semantics: In the context of a output-access, like getting a property value, or receiving the return of an operation call, the type can change *covariant* when the accessed feature is redefined. Since the redefinition can masquerade as the redefined, the redefining element can be access in any context the redefined can be accessed in. In the context of a input-access, like setting a property's value, or providing the arguments for an operation call, the type can change only *contra variant*; because only this way the redefinition can still masquerade as the redefined. In the example, p' can be access in any context that p is accessible. For properties that allow output- and input-access, like member variables, only *invariant* type changes are allowed.

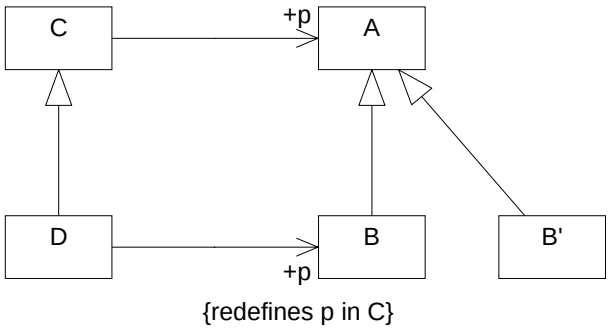

Figure 1. Redefinition example

We want to investigate what this diversity means for mapping the example given in figure 1. When a class C with property $p : A$ is specialized by class D with property $p' : B$, where p' redefines p and $A \lhd\!\!- B$, then according to the basic JMI mapping rules two interfaces are created. Interface `C` with methods `A C::getP()` and `void C::setP(A value)`, as well as interface `D extends C` with methods `B D::getP()` and `void D::setP(B value)` will be created on the Java side.

In Java terms: `B D::getP()` redefines `A C::getP()`, since it has the same signature. Fortunately Java supports covariant return types, and this piece of language mapping preserves the intended semantics. The set-methods, on the other hand, are problematic. In order for a method to redefine another method (or *override*; to overwrite is what it means to redefine in Java), the arguments of the redefining method must only change contra variant. But the argument of the update method changes covariant. Therefore `void D::setP(B value)` does only *overload* `void C::setP(A value)`. Unfortunately overloading does not have the semantics wanted.

Take the call `c.setP(aValue)`, where `c` is a reference of type C and `aValue` is of type B' with $A \lhd\!\!- B'$ but not necessarily $B \lhd\!\!- B'$. Since references are polymorph in Java, it is possible that `c` references a value of type D. What are the semantics of this call in this particular context?

The proxy object that implements the interface `D` has to implement both methods `void C::setP(A value)` and `void D::setP(B value)`. Since both methods simply overload each other (despite their shared name), the mentioned call `c.setP(aValue)` will invoke `void C::setP(A value)`. This is because `aValue` has type B', which is incompatible with B.

But this is not what MOF intends; instead we want $p' : B$ in type D to be updated, since it redefines $p : A$ in C. There is only one slot (value container) defined by the redefining

property p'. This slot is used for both properties p and p' because both describe actually the same property just with different types. The reason for the mappings failure is that we try to do a covariant type change on both, a output-access (`getP`) and a input-access (`setP`), on the same property. But static type safety can only be assured for a writing access with contra variant type change. In other words static type safety for redefining a property with both output- and input-access can only be assured for *invariant* type changes.

The solution to this problem is to postpone some type checking from compile time to run time. The proxy object's implementation of `void C::setP(A value)` must realize that it is redefined; it has to check whether `value` is of type B or not. If `value` is of type B, it delegates and calls `void D::setP(B value)`; if it is not of type B, it raises a type check exception. That way, retrieving a value can be type checked statically. To type check the setting a value is only possible against the most general type of a redefinition; complete type checking, however, is possible at runtime.

Redefinitions, where the redefining element has a different name, are another troubling point. The mapping leads to Java methods with different names and hence different signatures, and therefore there is no redefinition at Java level. We can apply a similar solution to solve this problem: The according method implementation realizes that the represented property has been redefined, it does the needed type checks, and delegates the call to the method that represents the redefining property.

For operations, MOF allows operation arguments to change covariant when redefined. This is a bit peculiar and the semantics are no further described. When operations with covariant arguments are mapped to Java, overloading semantics apply. Since overloaded methods are selected at compile-time it cannot be assured that the wanted operation is called. As before, the implementations of the corresponding Java methods have to decide at runtime with operation is to be invoked. [3] [4]

3.2. Multiple inheritance and merging

Even more challenging for the Java type system is the redefinition of multiple properties as they commonly occur when packages are merged. In the example, shown in figure 2, two classes with a property of same name, but different types, are merged. For instance, MOF used this several

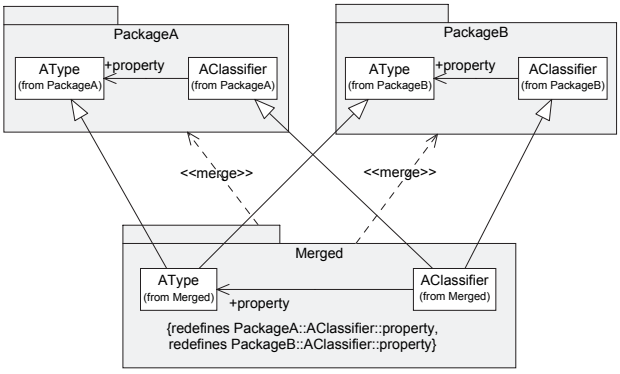

Figure 2. Merge example

times for the definition of the CMOF model. For example, the CMOF package is a merge of EMOF and UML infrastructure's Constructs.

In the example, the merge leads to multiple interface inheritance and a method that redefines two methods that are inherited from different super interfaces. The Java language does not forbid such a redefinition, but it does so, when the two inherited methods have incompatible return types. And since `PackageA::AType` and `PackageB::AType` are unrelated and therefore incompatible with each other, this mapping leads to faulty Java code. This holds true even when the redefining method has a return type, which is constructed to be compatible with both `PackageA::AType` and `PackageB::AType`.

The fact that such a mapping does not work in Java, does not necessarily mean that it cannot work for other static typed languages. From a type checking perspective the redefining method has a type that is covariant to both redefined methods and the problem lays in the too restrictive type system of the Java language.

For the Java mapping the only satisfying solution that we could find is to use the *combine* semantics described in MOF 2.0 Core. Combine is a special kind of merge, where the merging package contains all classes and features as it would with the regular merge, but all redefinitions and generalisation relations to the merged packages are omitted. This leads to a package with types that have all desired features, but that do not conform with the types of the merged packages.

3.3. Higher multiplicity

Properties can have arbitrary multiplicity; it means that they can represent a collection of values. Elements with such a multiplicity have to be mapped differently. For properties with higher multiplicity only one access method is generated with a Java type that allows to contain a collection of values. To update the values of the property, the

[3] This paper does not explain the runtime semantics of operations in detail. But, CMOF operation can basically be mapped on Java methods with implementations that the user has to provide, since the operation behaviour cannot be expressed with CMOF.

[4] Possible other semantics for operation parameter that could be worked into a language mapping, are *multi-methods* [14]. The implementation of such operations is resolved based on the runtime types of the arguments to on operation call.

collection retrieved through access method invocation has to be changed. In order to allow type safety and convenient programming that collection has to preserve the type of its elements. Java supports generics to parameterise collection types with concrete element types.

Those collections preserve type safety for both, output- and input-access to its elements. But this houses another problem: Two collection types $Set<A>$ and $Set<B>$ are not compatible to each other, even if $A \leftarrow B$. That means, if property $p' : B$ in D redefines $p : A$ in C and both property's have multiplicity that implies a collection of values then the mapping to `Set<A> C::getP()` and `Set<B> D::getP()` does not work, because the two return types are incompatible and would cause a compile time error. Generics are only compatible for invariant context type parameter due to the fact that a generic may use a context type parameter for a output- and a input-access.

For such cases Java allows to weaken the generics' context parameter: The reference type `Set<? extends B>` is covariant to `Set<? extends A>`. The context parameter is now unbound to covariant types. The negative side on unbounded context parameter is that every method that uses this context parameter as a type for one of its arguments (input-access), can not be called anymore. The reason is that the object beyond the reference of type `Set<? extends A>` could have a more restrictive context and therefore static type safety can not be assured anymore.

The solution to modify collections with unbound element type is the same as the solution for modifying redefined properties: The type checking is postponed until runtime. We implemented generic collection classes that use the most general Java type `Object` as element type for any method that modifies the collection, but a concrete type for all methods that return elements of a collection. That way, static type safety is assured for output-access on the collection' members, but modifying access can only be type checked dynamically.

4. Semantics for Associations and Property Sub-setting

The semantics for associations and property sub-setting in the CMOF model are explained in the MOF 2.0 Core [5] and the related recommendation UML 2.0 Infrastructure [6]. However, the models in those recommendations are only described as static constructs, and therefore their semantics are defined only for static models; the recommendations express semantics only in static conditions. But model repositories—basically dynamic programs—define operations to create model elements, operations to add values to the properties of an object, or remove values from the properties of an object.

When implementing such operations, non trivial questions arise that can not be answered by static constraints. The basic problem is: When the user makes a local change to a model, what else has to be changed in the model as a whole in order to let all constraints still be yield. For instance, when you add a value to a property that subsets other properties: How to change the model in a way that the subsetting constraints are still fulfilled?

We refine the semantics for associations and property sub-setting in this section. We propose a mechanism that uniformly includes semantics for associations and property sub-setting. This is reasonable, because associations and property sub-settings are two special cases for constraints between properties. Associations impose a bi-directional relationship between properties—namingly association ends; and property subsets imply that the values of one property always form a subset for the values of another property. Before we proceed to explain the semantics in a formal manner, we want to discuss our and other approaches to both subjects: associations and property sub-setting.

Associations have been used since the early works on object-oriented design; semantics and implementation strategies have been discussed widely. There are two general possibilities: (1) associations are classifiers in their own right with possible properties and specializations between associations; realized though association instances that link associated objects (Rumbaugh [27], MOF/JMI [5, 25, 21]); (2) associations are simply realized by unidirectional references with values that are constrained by the association (Graham, Bischof, Henderson-Sellars [18], EMF [11]). We used opportunity (2), because it defines an association as a constraint on the values of two corresponding properties. In MOF those properties are called association ends, which are basically properties of the association[5]). This allows us to handle associations in a similar way as we handle property subsets. However, realizing associations using constrained properties internally, does not mean that it is impossible to maintain the appearance of associations as classifiers and links as their instances.

Property sub-setting is a new feature in MOF and available in the CMOF-model. Amelunxen et al. [10] use the following approach: When adding a value to a property, it is also added to all subset properties and their subset properties and so on. This is straight forward; the values of properties remain subsets. When removing a value from a subset, remove it also from properties that are subsets to that property, concluding subsets remain subsets. We agree with the adding process, but the removing seems troubling. Imagine,

[5]The CMOF itself (the same for the UML or older MOF version) use the Rumbaugh way of modelling associations. In CMOF an association is a classifier. As each classifier, an association has member properties, but in addition an association has member ends, sub-setting member properties. These special properties associate classifiers (the property' types) with each other

adding a value to property p with subset property p' and deleting the same value from p again. It will be added to p', but not removed from p'. The remove operation does not completely reverse the add operation. This does contradict typical add/remove semantics.

We propose a different approach: Every time a value is add, we keep the information on which places the value is added (this information is later on called *set of depending slots*). Whenever this value is removed, it is also removed from all other properties that is was once added to. That way, we completely reverse the original intention of adding the value.

In the remainder of this section we define the semantics for adding and removing property values on a set theoretic basis. After we give some basic definitions, that relate relevant CMOF concepts to sets and relations, we introduce the notion of slots, and sets of depending slots. Finally we use those definitions to define semantics for adding values and removing values from object properties.

4.1. Properties, Slots, and Depending Slots

May P donate the set of all properties in a meta-model. We define for $p_k, p_l \in P$ the relation $p_k <_r p_l \Leftrightarrow$ p_k redefines p_l. It reflects redefinitions given in the meta-model. We define a reflexive, transitive hull for $<_r$ as $p_1 =_r p_2 \Leftrightarrow \exists p_1, \ldots, p_n : p_k <_r p_1 \ldots p_n <_r p_l \wedge p_l <_r p_1 \ldots p_n <_r p_k$. The equivalence classes of $=_r$ form slots. We define $slot(p) = \{p_i | p_i =_r p\}$ as a function that gives the corresponding slot to a property.

Later on we will use slots as container for property values. Because there is only one slot for all properties that redefine each other, the notion of slots has also redefinition semantics attached to it.

There are two relationships between slots, imposed by relationships between corresponding properties. First we define a relation for associations; for two slots s_1, s_2, $\rightarrow_a$ is defined as:

$$s_1 \rightarrow_a s_2 \quad \Leftrightarrow \exists p_1 \in s_1, p_2 \in s_2 : \\ p_1, p_2 \text{ are the members of an association}$$

Second is the definition of a relation for sub-setting; $\rightarrow_s$ is defined for two slots s_1, s_2 as:

$$s_1 \rightarrow_s s_2 \Leftrightarrow \exists p_1 \in s_1, p_2 \in s_2 : p_1 \text{ subsets } p_2$$

.

Based on those relations we construct a set of depending slots. Therefore, we define an update as $u = (o, v, s)$, where o denotes an object that contains slot s and v is the value that the slot s shall be updated (either added to the slot, or removed from the slot) with. To keep this simple we only consider object values here. For that matter v denotes always an object. We define a set of depending slots

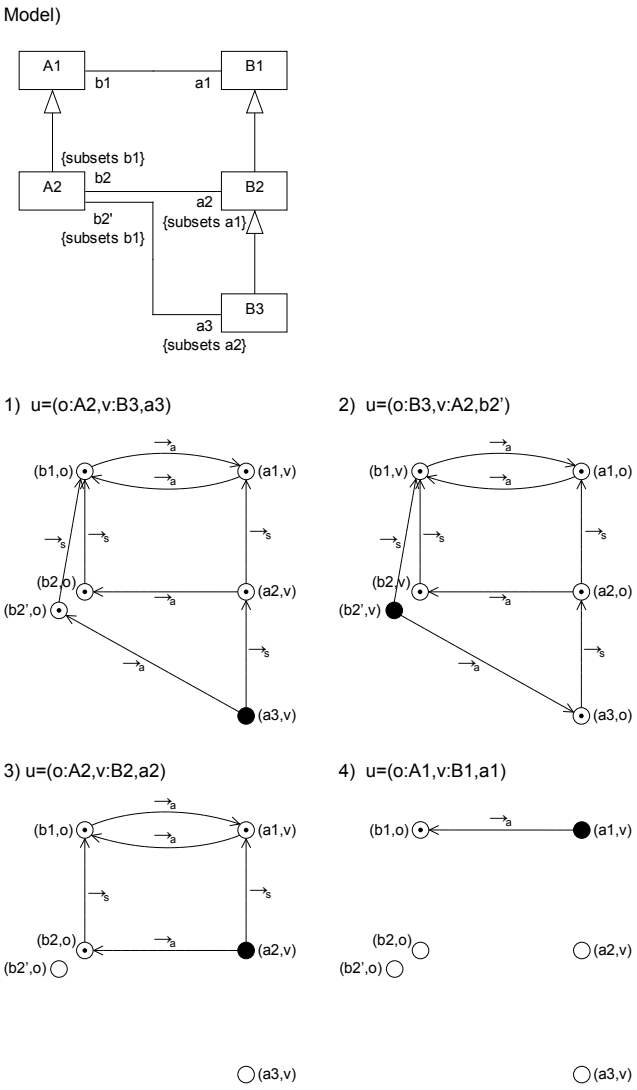

Figure 3. Update-set examples

regarding an update as a set of updates $ds(u)$. It is defined recursively; it contains the update u and all updates that are implied by associations and sub-settings for the properties in updates in $ds(u)$:

$$start: \quad (o, v, s) \in ds(u)$$

$$association: \quad \frac{(o_1, v_1, s_1) \in ds(u) \wedge s_1 \rightarrow_a s_2}{(v_1, o_1, s_2) \in ds(u)}$$

$$subset: \quad \frac{(o_1, v_1, s_1) \in ds(u) \wedge s_1 \rightarrow_s s_2}{(o_1, v_1, s_2) \in ds(u)}$$

Given the constrains in the UML-Infrastructure (that CMOF is based on), which regard validity of redefinition and subset contexts, $ds(u)$ can only contain

$(o_i, v_i, s_i), (o_j, v_j, s_j) \in ds(u)$ with $(o_i = o_j \wedge v_i = v_j) \vee (o_i = v_j \wedge v_i = o_j)$. In other words an update will only concern two objects, the object that is updated and the value that the object is updated with.

The example in figure 3 shows a model with five properties with different association and subset constraints between them. Below the model you see different examples of update sets. Dependencies between all slots are drawn as arrows; every dependent slots is marked as $\odot$; the originating slot that the update is initiated on, is marked as big bullet. The first example (1) shows the update set for value v (an object that has at least type $B3$) on object o (at least of type A2) for property $a3$. Sometimes there are several reasons why a slot depends on the originating slot; they have different arrows pointing at them. Because of the symmetry of $\rightarrow_a$ the update set for adding a value v (at least type $A2$ on object o (at least of type B) to property $b2'$ in example (2) contains the same slots as in example (1). Examples (3) and (4) show smaller updates where the properties $a2$ and $a1$ are updated.

4.2. Add and Remove Values to Properties

Meta-modelling frameworks use operations on models to modify them. These can be object creation, deletion, or the adding and removing of values to and from properties. May Op be a set of such operations. We only want to give semantics for $\texttt{add}(o, v, p) \in Op$ and $\texttt{remove}(o, v, p) \in Op$. These are parameterised operations that add or remove a value v in property p of object o. May E denote the state of an model. $E = op_1 \circ \ldots \circ op_n$ results from successively calling operations $op_i \in Op$. For each E and object o, $f_{o,E}(p)$ denotes the set of values that the property p of object o has in state E. We use these function in predicates $P(f_{o_1,E}, \ldots, f_{o_n,E})$. Alternatively we write $E \models P(f_{o_i}, \ldots, f_{o_n})$ to denote that a state E yields predicate P for the objects, properties, and their values in E.

The predicates $AllSet_{o,v,p}$ and $NonSet_{o,v,p}$ describe a model state, where all depending slots of update (o, u, p) contain the according value, and a model state where all those slots do not contain those values.

$$\begin{aligned} AllSet_{o,v,p} &:= \forall (o_i, v_i, s_i) \in ds\left((o, v, slot\left(p\right))\right) : \\ &\quad \forall p_i \in s_i : v_i \in f_{o_i}(p_i) \\ NonSet_{o,v,p} &:= \forall (o_i, v_i, s_i) \in ds\left((o, v, slot\left(p\right))\right) : \\ &\quad \forall p_i \in s_i : v_i \notin f_{o_i}(p_i) \end{aligned}$$

Using those predicates, the following rules describe `add` and `remove`:

$$\frac{E \models NonSet_{o,v,p}}{E \circ \texttt{add}(o, v, p) \models AllSet_{o,v,p}}$$

$$\frac{E \models NonSet_{o,v,p} \wedge E \circ op_1 \ldots op_n \models NonSet_{o,v,p}}{E \circ \texttt{add}(o, v, p) \circ op_1 \ldots op_n \circ \texttt{remove}(o, v, p) \models NonSet_{o,v,p}}$$

When adding values to a property, all depending properties are updated too. This is straightforward, opposite association ends are assigned accordingly and subset properties are updated to remain supersets. The remove, however, might be more peculiar. When removing a value, it is also deleted from the set of depending slots that was originally used to add the value. This especially means that for all $p_1, \ldots, p_n$ and $(o_i, v_i, s_i), (o_j, v_j, s_j)$ with $p_i \in s_i, p_j \in s_j$ and $s_i, s_j \in ds((o, v, slot(p)))$:

$$\frac{E \circ \texttt{add}(o, v, p) \circ \texttt{remove}(o_i, v_i, p_i) \models A}{E \circ \texttt{add}(o, v, p) \circ \texttt{remove}(o_j, v_j, p_j) \models A}$$

In other words, no matter on what slot of depending slots of (o, v, p) we remove a previously added value from, all slots that were originally updated when adding the value, are considered. That way, all values are removed that are assigned to the original intention of adding a value. Example 1 in figure 3 for example: No matter whether o is removed from $b1$, $b2$, or $b2'$ or v is removed from $a1$, $a2$, or $a3$, they are all removed, because the reason for each value is in each slot that v has been added to $a3$. The same holds for the other examples.

The rules given here are far from complete. Unfortunately it is not possible to give all rules, considering all possible operations, within the scope of this paper. A lot of non-trivial cases arise especially in the context of certain property features, like ordered values, uniqueness of values, multiplicities, non object-values, etc.

5. Conclusions and Future Work

In this paper the language mapping and semantics needed for a MOF 2.0 implementation based on the CMOF model were presented. UML 2.0 and MOF 2.0 allow the redefinition of classifier features in the context of a generalisation; this advantage can be preserved for the Java mapping by using covariant return types. Further more can generic collections types be used for a type safe access to values of properties with higher multiplicity. Compared to the programming practice with MOF 1.x and JMI better abstract definitions and programming with out heavy usage of type casts and reflection are possible. Only the modifying access to redefined properties can not relayed to the native static Java type checking, and it has to be done at runtime. The presented semantics for associations and the sub-setting of properties allow implementations for automatic update of depending properties when the values of a property are changed. The presented work was implemented in a Java modelling framework called *A MOF 2 for Java*.

This MOF 2 implementation transfers all the possibilities for modularisation and reuse that MOF 2 offers at modelling level, to the implementation level. Since CMOF is the basis for UML, a CMOF implementation should be the basis for UML 2 tools and for tools using languages based on UML 2, i.e. UML-profiles. Besides using CMOF for UML and UML-profiles, CMOF also presents a valuable modelling basic for others. For instance, modularisation and reuse, combined with tool support, could be a strong foundation for domain specific languages (DSLs). Those languages depend on ad-hoc tools support, and for the development of DSLs it is essential to derive tools from model-based language definitions easily and fast.

Our group uses the *A MOF 2 for Java* for the development of language tools for integrated software engineering with UML and ITU-languages (SDL [19] and eODL [20]) [13]. We research different approaches, based on reuse of UML's infrastructure in meta-models for ITU-languages and based on UML profiles for ITU-languages. We are currently enhancing our modelling framework by implementations of auxiliary operations for UML's infrastructure library. Those operation convey much of the static semantics of the infrastructure concepts. This in addition with an integration of OCL tools, could be used for expressing and actually checking the static semantics of models based on UML's infrastructure. For better support for UML profiles, we are implementing a mechanism that allows repository generation for this special kind (UML-profiles) of meta-models.

Besides advancing the modelling of static aspects, CMOF can also improve the definitions of semantics, especially in terms of translational and operational semantics: The success of MDA depends on automated model transformations. OMG's request for proposal for a MOF transformation language [7, 26] resulted in proposed languages that use meta-model fragments for selecting transformation rules. Rules that therefore depends on the expressiveness of the meta-modelling language (i.e. CMOF). Of course, transformation engines, based on those proposals, depend on implementations for CMOF. Operational semantics can easily be achieved by defining operations in meta-models and implementing them in a programming language, using a meta-modelling framework with an according language mapping. Conducting such operational semantics is simpler, when the meta-modelling language is more expressive and this expressiveness is realized in the programming environment, such as in static types and automated update of subset properties.

With MOF's CMOF model meta-modelling becomes more expressive, and meta-models more reusable. This paper showed how CMOF concepts can be realized in programming platforms and therefore used for the development of model based tools for engineering of complex computer-based systems.

References

[1] *A MOF 2.0 for Java*. Lehrstuhl für Systemanalyse, Humboldt-Universität zu Berlin.

[2] *ModFact (Model Factory)*. Object Web – Open Source Middleware.

[3] *Netbeans Meta Data Repository (MDR)*. NetBeans / Sun Microsystems.

[4] *CORBA Component Model, Version 3.0 - CIF Metamodel*. Object Management Group, June 2002. formal/2002-06-77.

[5] *Meta Object Facility (MOF) 2.0 Core Specification*. Object Management Group, Oct. 2003. pct/03-10-04.

[6] *UML 2.0 Infrastructure Specification*. Object Management Group, Sept. 2003. pct/03-09-15.

[7] *MOF 2.0 Query/Views/Transformations RFP*. Object Management Group, Apr. 2004. ad/2002-04-10, Request for Proposal.

[8] *UML 2.0 Superstructure Specification.* Object Management Group, Oct. 2004. pct/04-10-02.

[9] Adaptive, Compuware Corp., Interactive Objects, Sun Mircosystems. *MOF 2.0 Facility and Object Lifecycle Specification*. Object Management Group, Apr. 2004. ad/04-04-02, Joint Revised Submission.

[10] C. Amelunxen, A. Schuer, and L. Bichler. Codegenerierung fuer Assoziationen in MOF 2.0. In *Modellierung 2004*. Gesellschaft fuer Informatik, 2004.

[11] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework (The Eclipse Series)*. Addison-Wesley Professional, Aug. 2003.

[12] T. Clark, A. Evans, P. Sammut, and J. Willans. *Applied Metamodelling, A Foundation for Language Driven Development*. 2004.

[13] J. Fischer, A. Kunert, M. Piefel, and M. Scheidgen. ULF-Ware – An Open Framework for Integrated Tools for ITU-T Languages. In *Twelfth SDL Forum*, June 2005.

[14] R. Forax, E. Duris, and G. Roussel. Java multi-method framework, 2000. In International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00), November 2000.

[15] Fraunhofer Institute FOKUS. *MOF 2.0 to OMG IDL Mapping*. Object Management Group, Jan. 2004. ad/04-01-16, 2nd Revised Submission.

[16] A. Gerber and K. Raymond. MOF to EMF: There and Back Again. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[17] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification Third Edition.* Addison Wesley, 2005.

[18] I. Graham, J. Bischof, and B. Henderson-Sellers. Association considered a bad thing. In *Journal of Object-Oriented Programming*, Feb. 1997. Vol. 9, no. 9, pp. 41-48.

[19] ITU-T Z.100. *Specification and Description Language (SDL)*. International Telecommunication Union, Aug. 2002.

[20] ITU-T Z.130. *Extended Object Definition Language (eODL)*. International Telecommunication Union, July 2003.

[21] JMI. *The Java Metadata Interface(JMI) Specification(Final Release)*. Java Community Process, June 2002. JSR-000040.

[22] D. Karagiannis and H. Kuehn. Metamodelling Platforms. In *Proceedings of the Third International Conference EC-Web*. Springer-Verlag, Berlin, Heidelberg, 2002. p. 182.

[23] Kim B. Bruce. *Foundations of Object-Oriented Langauges: Types and Semantics*. MIT Press, 2002.

[24] MDA. *Model Driven Architecture Guide, Version 1.0.1*. Object Management Group, June 2003. omg/03-06-01.

[25] MOF. *Meta Object Facility, Version 1.4*. Object Management Group, Mar. 2003. formal/2002-04-03.

[26] QVT-Merge Group. *MOF 2.0 Query/Views/Transformations*. Object Management Group, Apr. 2004. ad/04-04-01, Revised Submission.

[27] J. R. Rumbaugh, M. R. Blaha, W. Lorensen, F. Eddy, and W. Premerlani. *Object-Oriented Modeling and Design*. Prentice Hall, 1st edition, Oct. 1990.

[28] XMI. *XML Metadata Interchange, Version 1.2*. Object Management Group, Jan. 2002. formal/2002-01-01.