On Implementing MOF 2.0 – New Features for Modelling Language Abstractions

Markus Scheidgen

Humboldt Universität zu Berlin
Institut für Informatik
Unter den Linden 6
10099 Berlin, Germany
scheidge@informatik.hu-berlin.de

Abstract. The 2.0 series of OMG MDA standards offers a set of new features, which accommodate the tool-based development of languages with better means for modelling abstractions and for the reuse of common language elements. We choose to implement the new MOF 2.0 Core recommendation as modelling facility for our model driven engineering tools. This paper is about the problems, experiences, and solutions encountered during the implementation of a MOF tool, based on the CMOF model. The used design rationale is a lightweight meta-modelling repository technology that focuses on meta-modelling with abstractions and common concepts.
Therefore special attention is given to the realisation of UML-Infrastructure's new abstraction and generalisation features, especially property subsetting, redefinitions, and merges, as well as a language mapping that carries most of these infrastructure features into the target language, for a safe and convenient programming with models.
The topics that the paper covers are a MOF to Java mapping, and especially the problems that occur when MOF constructs are represented within the rather strict static type systems of programming languages; a discussion of property subsetting semantics and its implementation; other usability, performance, and general architecture issues.

1 Introduction

It is nearly a decade ago that the first major release of UML brought a unified graphical language for high-level software modelling to the software engineering community. Since that times, ways have been found to adequately define and machine handle models and specifications through meta-modelling. Now, close to the finalisation of the version 2.0 instalments of the modelling specification recommendations for UML [1] and MOF [2], languages can be described with highly decomposed language concepts, multiple layers of abstractions, and reuse of common language elements. The UML and MOF specifications themselves display the use of those techniques: UML's Infrastructure defines abstract, common language concepts that are reused throughout MOF and UML's superstructure [3].

We intend to go a step further and use common, abstract language concepts for the definition, alignment, and tool based development of multiple modelling and programming languages. In [4, 5] we propose a meta-model based methodology for the creation of tools for the different tasks of model driven development. Those tasks are model transformation, static analysis, code generation, etc. The overall goal is to create MDA tool chains that use for each abstraction level and each system view the language that is most adequate for the task.

For this endeavour, as well as for the growing number of similar projects, emerging with the increasing popularity of MDA, a technology is needed that allows flexible meta-modelling with enhanced support for modelling with abstractions. MOF 2.0 in combination with the UML 2.0 Infrastructure represents this very technology, providing new features for model element redefinition, generalisation, and infrastructure libraries, which already specify abstractions and common constructs. Therefore we choose to implement a MOF 2.0 repository.

Our goal is a lightweight MOF implementation with an easy to program API and a language mapping that allows convenient and safe handling of models in all layers. This paper is about realising the MOF 2.0 Core, the problems we had, the solutions we choose, and experiences we made, with special focus on features that are new to MOF 2.0.

The section following to this introduction explains the requirements for our implementations, based on our needs for a meta-data repository. It also gives an introduction to the architecture and design rationale. After clarification about the general mechanisms, section 3 gives a more detailed inside into the language mapping we choose, the problems that occur when the very flexible MOF model is mapped to the more strict type system of a programming language, and how our mapping contrast to the MOF 2.0 IDL mapping [6] that is recommended by the OMG. The following section 4 discusses implementation strategies and performance issues for link consistency and property subsetting. Other issues are briefly addressed afterwards; these are: How we integrate user code into generated model repositories, and how performance can be increased by using static meta-models. The conclusions section gives a short summary and an outlook about integrating other technologies like OCL [7] and MOF 2.0 QVT [8].

2 Overview

2.1 Rationale and Requirements

The meta-modelling facility we need has to fulfil a collection of specific requirements. First, we want a lightweight implementation; a lightweight implementation characterizes a small MOF that is reduced to the features essentially needed to work with meta-data. This includes handling of meta-models, generation of repositories from those meta-models, XMI import export facilities, and reflection facilities. What we do not need are full scale database persistency, transaction safety, event mechanisms, remote access, use of distributed middleware, etc. But of course the tool should designed in a way that additional functionality can be integrated later.

Second, we had to decide which part of MOF we want to implement. We choose to support meta-modelling with the CMOF model, because this is the language that contains the expressiveness that characterises MOF 2.0 [2] and that will give us the most benefit. The EMOF model in contrast does only cover the means of MOF 1.x.

Third, the implementation will be used for the development of other tools written in the targeted implementation language, Java. Therefore, it has to provide a meta-data interface that can be programmed conveniently and type safe. In our experience MOF 1.x implementations have always lead into unsafe and annoying coding practices with lots and lots of type casts, and unnecessary indirections.

Here is a list of the most important requirements:

- Implementation of the CMOF model.
- Implementation of the UML Infrastructure library. This library has to be used as a basis for the CMOF model, as well as it has to be useable in user meta-models.
- XMI import and export of meta-models and models. This is the only persistence and model exchange facility needed.
- Implementation of the recommended instance model. Special emphasis lays on the exact implementation of redefinition and property subsetting.
- Implementation of reflection facilities.
- Code generation for interfaces and object proxy implementations that use the instance model.
- It must be possible for the user to provide custom code for derived features and operations.
- The handled meta-data has to be accessible through a convenient and type safe API.

2.2 Architecture

The architecture of our implementation is shown in figure 2.2. The heart of this implementation is the *instance model*. It is the central data structure that stores and controls all model elements. It keeps meta-data independent of its meta-meta, meta, or simple model nature; it is independent from the modelling layer. This *instance model* is suggested by the MOF recommendation and its key responsibility is to keep model elements and hold them in a consistent state. The instance model is used by the *reflection API*. This reflection facilities provide an untyped interface to model elements stored in the instance model. The reflection API can directly be used by the user; but it is suggested to use the more safe and convenient access through generated *repositories*. Those repositories provide a meta-model depended and type safe interface access to the model storage. A repository for CMOF is initially provided, user repositories can be generated based on CMOF meta-models provided by the user. The XMI import and export services are based on the reflection API and thus work meta-model independent.

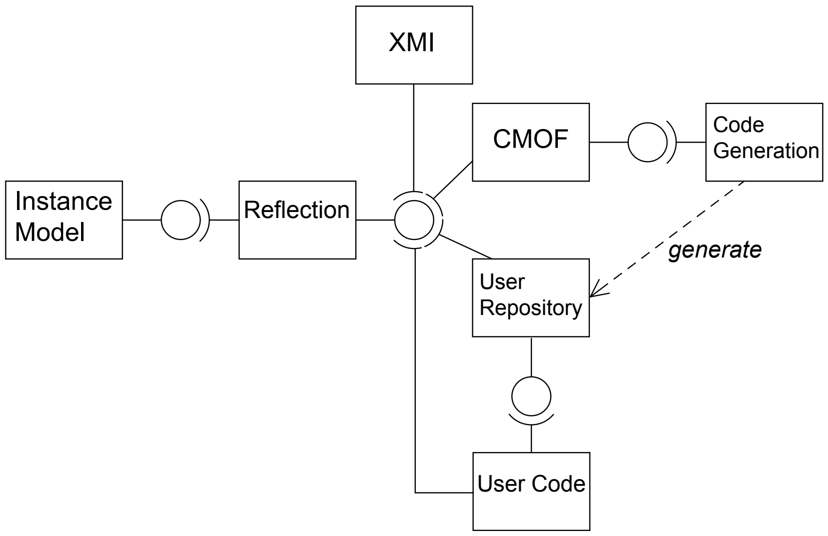

Fig. 1. Architecture

The instance model stores model elements as native objects in respect to a defining meta-classifier. Those objects have value slots for each property that is defined in the meta-classifier. Slots can be filled with value specifications that represent other instances, structured data-values, or primitive data, according to the type of the defining property. It is the instance model's responsibility to ensure that model elements comply to constraints implied by the definitions that are given by the according meta-classifier.

The reflection API offers three facilities: Objects, Factories, and Extents. An object covers an instance from the instance model; it provides access to its properties, dispatches calls for operations and derived features to user code, and allows instance deletion. Those objects are called *proxy objects*. Factories offer functionality to create instances. Extents structure the model space into distinct living spaces for instances. Each of this extends reflect a distinct meta-model instance, and allows to collect the elements of a model by their meta-type. It is the reflection's responsibility to control objects and their lifecycle.

Repositories consist of simple generated classes and interfaces that work as a typed wrapper for reflection objects. For each meta-classifier code is generated that allows to access the underlying instances in a type safe and convenient way by using the reflection facilities. Those repository code can be generated automatically from meta-models written in CMOF. A repository consists of generated meta-model specific interfaces for proxy-objects, implementations of proxy-objects, and model factories.

3 Language mapping

What is a *Language mapping*? The concepts of the language that is used for meta-modelling, CMOF, have to be related to the concepts of a target language, in a way that equal semantics can be achieved. Therefore to give a language mapping means: Tell how every single CMOF concept is realised in the target programming language.

The target language: We choose to implement MOF for the Java programming language. The reasons for that decision are to manifold to be explained here, but we want at least reason, why we did not choose the obvious and by the OMG recommended IDL. Compared with IDL, Java provides two important features that IDL does not: *Covariant* return types, that were introduced with Java Platform 2 Version 1.4 and *generics* (a concept known to the C++ community known as templates), introduced with Java Platform 2 Version 5. Those features will prove necessary for a convenient modelling API. For further explanations the mapping has to be introduced first; the importance of covariant return types and generics will be discussed later in this section. For more information on type checking and static semantics refer to [9, 10].

We plan to use the MOF for implementing tools directly in Java. This means that we need the MOF accessible through an usable, convenient API. For the language mapping this leads to the following rationale: 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 must not be a part of the MOF programming routine.

3.1 The language mapping in general

The previous MOF recommendation MOF 1.x [11] has a Java mapping defined in the *Java Metadata Interface(JMI)* [12]. The JMI has already constituted the common fashion of a MOF to Java mapping. From a superficial view that mapping realises the modellers intuition, when he looks on a meta-model notated in a UML class-diagram: A MOF elements are represented through Java objects, MOF classes are mapping to Java interfaces, , a MOF method to a Java method, a MOF attribute to a Java property, etc. The more basic mapping rules are listed in figure 2.

Every model element is represented through a proxy object, that implements the interface that corresponds to the according meta-class. To call operations on that model element, or to access its attributes, or linked objects, the according Java methods have to be invoked. For more information about MOF semantics and language mappings refer to the old *MOF 1.4* [11] and *JMI* [12] or the new *MOF2.0 Facility and Object lifecycle* [13] and *MOF2.0 IDL* [6] recommendations. The rest of this section will handle the detailed mapping of properties and operations in the context of redefinitions, merges, and arbitrary multiplicity.

MOF concept	Java concept
class	interface
specialized class	extended interface
classifier scope feature	static method
instance scope feature	static method
operation	method
attribute and navigable association ends	pair of get- and set-methods

Fig. 2. Some basic rules for the MOF to Java mapping

3.2 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 for instance, 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 most important for the language mapping. Similar constraints apply to the redefinition of operations; most important, argument types and return type must change covariant.

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 $A \triangleleft\!\!- B$. This is a *covariant type change*. A operation $o' : B_1, \ldots, B_n \rightarrow B_{return}$ can redefine $o : A_1, \ldots, A_n \rightarrow A_{return}$, if $A_i \triangleleft\!\!- B_i, A_{return} \triangleleft\!\!- B_{return}$.

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++, IDL. Most of those languages have the following, or even more restrictive redefinition semantics: In the context of a read 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 write 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 and o' can be invoked in any context and for any arguments that o can be invoked with. For properties that allow read and write access, like member variables, only *invariant* type changes are allowed.

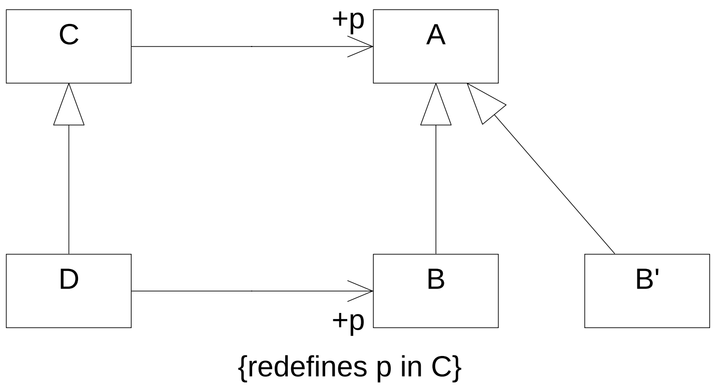

Fig. 3. Redefinition example

We want to investigate what this diversity means for mapping the example given in figure 3.2. 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 mapping rules in figure 2 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 update methods, on the other hand, are problematic. In order for a method to redefine another method (or *override*, that 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. It is the *defining property* for both properties. The reason for this failure is, that we try to do a covariant type change on both, a reading access (`getP`) and a writing 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 read and write 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 realise 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 an exception.

Another troubling point are redefinitions, where the redefining element has a different name. 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 realises 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. The UML infrastructure says that *to augment* is a possible redefinition semantics; we choose to stick to the overloading semantics provided by Java, and interpret them as *augmentation*.

3.3 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 3.3, two classes with a property of same name, but different types, are merged. This happens several times in the definition of the MOF model. Take the CMOF package; it 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 can not 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

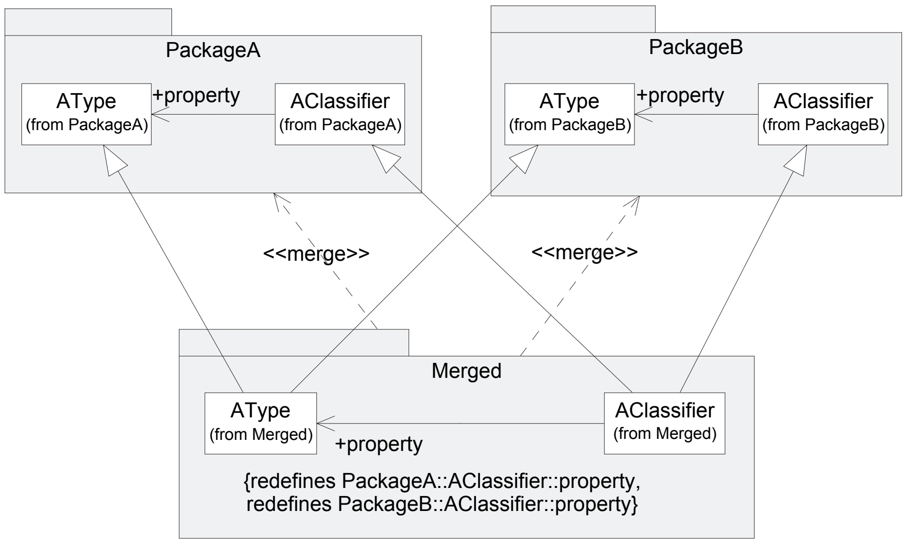

Fig. 4. Merge example

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.

3.4 Higher multiplicity

Properties can have arbitrary multiplicity, what 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 a access method is generated with a Java type that allows to contain a collection of values. To update the values of the property, the 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 (in C++ a similar concept is called templates) to parameterise collection types with concrete element types.

Those collections preserve type safety for both, reading and modifying 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.

For such cases Java allows to weaken the generic's 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, 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. That way static type safety is assured for reading access on the collection' members, but modifying access can only be type checked dynamically.

3.5 IDL

To close this section, we want to give a short comparison to the mapping recommended in MOF IDL [6]. Similar mapping rules for properties and operations apply for IDL. For redefinitions no new access methods for the redefined property are created, due to the fact that IDL does not support covariant return types. This leads to necessary type casts in contexts that use the specialized type of a redefined property. Elements with higher multiplicity use collection types that are generated for each possible element type. That way type safety can be ensured even in the absence of a native IDL template concept.

4 Links, References, and Subsets

4.1 Associations and reference attributes

In MOF associations are used to model concept relations. Those associations are instantiated by links meta-model instances. MOF associations are binary relations, relating always two classifiers with each other. The two ends of an associations are modelled with properties, called association ends. These properties have the related classifiers as their types. Association ends can be navigable. The property representing a navigable end is owned by the classifier on the other side of the association. This classifier can navigate over the link by accessing this association end property. We will call those ends, owned by classifiers, *reference attributes*.

For reference attributes different implementation strategies can be used: (1) store only reference attribute values and derive link information; (2) only store information about links and derive reference attribute values from link data; (3) store both links and reference attribute values and hold both consistent.

The third (3) strategy would lead to unnecessary resource usage of space and time. Strategy (2) seems to be inadequate. From our experience, links are usually not used directly. Links are of rather limited use. It is more common to navigate from object to object by accessing reference attributes. The same for link creation: Setting the values of a reference attribute is simpler and more intuitive. The remaining strategy (1) has the disadvantage that it would need implementation for both, deriving links from reference attributes (for navigable ends) and explicitly storing link data (for non-navigable ends). To achieve a simpler design, we create non-visible reference attributes for all non-navigable ends that are owned by the relating classifiers. This way no link storage is necessary because all link information lays in reference attribute values and can be derived.

An association creates an dependency between two reference attributes. The semantics are that when a reference attribute on `instanceA` is assigned a new value `instanceB`, the opposite property on `instanceB` has to be updated to reference `instanceA` too. Associations create update dependencies between properties.

4.2 Property subsetting

MOF/UML 2.0 allows properties to *subset* other properties. The semantics are that the values of a subsetting property are always a subset of the values of the subsetted property. Subsetting may be done across the borders of the defining classifier, but only along generalisations. The context of a subsetting property (the owning classifiers for both, reference and regular owned attributes) has to conform, e.g. specialise, the context of the subsetted context. Property subsetting imposes another kind of dependencies between properties.

To implement those subsetting semantics different implementation strategies are possible: (1) the values for all properties are stored as usual and hold consistent according to their subsetting relationships; (2) only supersets are stored and subsets are derived; (3) only subsets are stored and supersets are derived.

Strategy (3) is wrong and results in faulty semantics. It would be impossible for a superset to contain values that are not contained in one of the subsets. Actually, this derived superset is a special property characteristic called *derived union*. The third strategy (3) leads to serious questions: The only way to derive a subset from a superset is to constrain the superset values with the subsetting property's type (this is one possibility recommended in the MOF 2.0 Core). But what if two properties have the same type; what if class `Container` has properties `Member` and `OwnedMember` {`subsets Member`} both of type `Element`. If only `Member` is stored, how can `OwnedMember` have different values then `Member`.

Unfortunately even the remaining strategy (1) leads to open questions. For example, does removing an element from the values of an subset also mean to remove it from all supersets? However, solution (1) seems to be the most obvious, and it provides more flexibility to alter the semantics later in order to fit recommendation changes, distinct variation points, etc. We choose to implement strategy (1): The values of the properties are stored for themselves, but subsetting creates update dependencies between properties of the same object. The fact that two properties that subset each other, are instantiated in the same object results from their conforming contexts.

4.3 Update graphs

Both, associations and subsettings, result in dependencies between the properties, that cause dependencies between the instances of properties, the attributes of objects and their values. When the value of an attribute in an object is changed, this also implies changes to the values of depending attributes in the same (for subsetting) or in other objects (for associations). To collect all instances of depending properties, we use a helper construct, called *update graph*.

But first, we need some definitions. Consider that the values of an attribute are changed. The instance that owns the attribute is called *owning instance*. All changes can be reduce to adding or removing a single change value from the set of values that the attribute represents. If the change value is another instance, it is called the *opposite instance*. If the change value is not another instance, but just a data value, it can be ignored, because its classifier cant be associated. The owning instance and the opposite instance form the *change context*.

Values of attributes are stored in *slots*. A slot has a *slot context*. This is the classifier of the instance that owns the attribute that represents the slot. A slot represents an instance of a property that has no further redefining properties in the slots context; this property is called the *defining properties* of that slot. The slot also represents the instances of all properties that are directly or indirectly redefined by the defining property.

Dependencies between properties, imply dependencies between slots, so that when the values in a slot change then the values of depending slots in the owning instance as well as in the opposite instance may change too. A slot s depends on another slot p if:

- s is owned by the owning instance, p is owned by the opposite instance, and the defining property of p or one of its redefined properties is the opposite property to the defining property of s or one of its redefined properties (association).
- p's defining property or one of its redefined properties is a subsetted property of s's defining property or one of its redefined properties (subsetting).

In order to compute which slots have to be updated when a property is changed for a specific update context, we use the *update graph* of the slot to be updated. The update graph is a directed graph; it is constructed as follows:

- In the beginning, the graph only consists of the slot to be updated
- For every slot in the graph all depending slots are also in the graph
- If slot s depends on slot p an edge from one to two is part of the graph.

Calculation of the update graph is straightforward; with it, the necessary updates can be archived easily.

Update graphs depend on the update context, which is a dynamic parameter and therefore update graphs can not be calculated in advance. They have to be computed for every update context; this can be a major performance issue in many cases. However if performance is an important factor, updated graphs can be computed statically under the assumption that no polymorphy used.

The problem is that subsettings and associations are defined on static and often abstract types. But, the actual *dynamic* context that a attribute or link is accessed in, may differ from the *static* context that the defining property is defined in. In the dynamic context the property might be redefined and thus augmented with additional subsettings or associations. The ability to masquerade values of different concrete types as value of a more abstract type is commonly called polymorphy.

To abandon polymorphy in the context of properties means to disallow that a redefining property adds characteristics that distinguishes it from the redefined property other then a covariant type. Therefore a redefining property must not add any subsetted properties, and a redefining property must not be associated with a property that is not a redefining property to properties that are already associated with the original property. Under these circumstances update graphs can be computed based on properties rather then the slots that represent their instances. Those *static* update graphs can be build at compile time.

We use the update graph to defined the following semantics for properties: When ever a value is added to a property, an according update graph is calculated, and the value is added to all the corresponding nodes (subsetted properties and opposite association ends). The update graph is then saved, as a property of all the values added to all the nodes of the update graph. The reason for that is to keep the update graph as a reason, an agenda for every value. When you now remove a value, or more general change a value, all the values in the hole update graph are touched, no matter on what node the change was initiated on. That way all values will be updated, that either depend on the updated value, or that are a reason for the updated value's existents.

5 Other issues

5.1 Integration of user code for derived properties and operations

Properties in MOF can be signed *derived* in order to specify that this property's value is determined by other model elements. For those properties the user has to provide code that computes the values. Operations have also no predefined behaviour and have to be specified by user code.

In order to provide such code, it is the user's responsibility to implement access and update methods (in the case of a derived property) or the representing method (in case of an operation). However, since the language mapping depends on the multiple inheritance of Java interfaces, providing implementations, using Java classes with single inheritance, is not trivial. The steps that have to be done by a user to provide his code, should be as transparent and intuitive as possible.

We implemented the delegator pattern to achieve multiple inheritance for user implementations. For every generated MOF classifier, thus for every generated Java interface, a special delegator class is generated that implements the according interface. All non derived features are implemented as normal proxy object implementations, accessing the underlying instance model. All derived features are implemented as empty methods or methods return the *null* value.

The user can now extend delegator classes and add additional implementations. Since the delegator is implemented as a normal proxy object, all properties can be access by the user code via the `this` reference; the user has not to care about special delegator semantics.

When now an MOF classifier is instantiated, the Java class loader is used to look for user delegators that correspond to the MOF classifier or one of its generalisations. Any suitable delegator found is registered as a delegator of the proxy object that represents the instance.

Now, whenever a derived property or operation is accessed, the proxy object uses the Java reflection API to look for user methods in all registered delegators. When no implementation is found an exception is raised. When multiple implementations are discovered, one is selected according to redefinition semantics. The chosen implementation is finally invoked.

5.2 Enhancing performance with static meta-models

In section 2.2 the instance model was introduced. It stores all instances and the values of their attributes; it does this in respect to the definitions given by a meta-classifier. This means that the instance model has to reflect on the properties of the meta-class, the properties of the properties, etc. Since the meta-model itself is just a model and hence handled by the instance model, the access of meta-data also involves reflection on the meta-meta-class, its properties, and properties of those properties. A simple element access or update may cause a lot of computational work; this behaviour can lead to serious performance problems.

To improve the performance, we utilize the fact that after a meta-model is instantiated, it will not change; a meta-model that is completely developed and is only used for instantiation will not change anymore. Therefore it is not necessary to store the meta-model in the instance model, with all its consistency, type, meta-meta-dependent hooks and checks. Instead we use a serialized and static version of meta-models for instantiation. Such a *static meta-model* is not stored in the instance model, it is directly implemented with special versions of proxy objects. These proxy objects hold the values to all properties as constant, primitive Java values. Accessing properties simply leads to returning a constant. Reflection on meta-elements can thus be done in small and constant time.

6 Conclusions

There are still weaknesses and we will keep improving our tool. One point of special interest is the navigation in models and querying model elements. As in the last MOF 1.x the native mechanisms for querying models in general are weak. MOF 2.0 does not even support reflection on composites and components; thus it is even harder to navigate a model in a meta-model independent way. But it is intended by the MOF 2.0 QVT recommendation that OCL 2.0 should be used as a query language. Therefore we plan to integrate an OCL 2.0 implementation, not only for evaluating constraints but to also as a query language.

OCL is very powerful and expressive but it also has the weakness to be a heavy machinery. Often a simple API is needed that allows very simple queries, or queries that are computed at runtime. Therefore a small, but perhaps not that expressive, query language should be implemented. Such a facility could be part of an extended MOF reflection API. Another project, developed in our department, is MOPA. MOPA is a tool for navigating tree like data structures with typed nodes. We already used MOPA successfully to navigate and transform MOF 1.x based models, utilizing the fact that the composite-component relation induces trees in models.

In this paper we presented a MOF 2.0 Core implementation based on the CMOF model. UML 2.0 and MOF 2.0 allow the redefinition of classifier features in the context of a generalisation; this advantage could be preserved for the Java mapping. Further more allow generic collections types type safe access of 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 is possible. Only the modifying access to redefined properties could not relayed to the native static Java type checking, and it has to be done at runtime. Redefinition and subsetting is realised in a transparent way, preserving the essential polymorphism. The presented MOF tools allows the user to integrate his own code for derived features and operations. In conclusion programming with meta data is more convenient and safer, even or just because of the increased expressive power of the MOF model.

References

1. UML: UML 2.0 Infrastructure Specification. Object Management Group (2003) pct/03-09-15.
2. MOF: Meta Object Facility (MOF) 2.0 Core Specification. Object Management Group (2003) pct/03-10-04.
3. UML: UML 2.0 Superstructure Specification. Object Management Group (2004) pct/04-10-02.
4. Fischer, J., Holz, E., Prinz, A., Scheidgen, M.: Tool-based Language Development. In: Workshop on Integrated-reliability with Telecommunications and UML Languages. (2004)
5. Fischer, J., Andreas Kunert, M.P., Scheidgen, M.: ULF-Ware – An Open Framework for Integrated Tools for ITU-T Languages. In: Twelfth SDL Forum. (2005)
6. Fraunhofer Institute FOKUS: MOF 2.0 to OMG IDL Mapping. Object Management Group (2004) ad/04-01-16, 2nd Revised Submission.
7. UML: UML 2.0 OCL Specification. Object Management Group (2003) ptc/03-10-14.
8. QVT-Merge Group: MOF 2.0 Core Query/Views/Transformations. Object Management Group (2004) ad/04-04-01, Revised Submission.
9. Kim B. Bruce: Foundations of Object-Oriented Langauges: Types and Semantics. MIT Press (2002)

10. Gosling, J., Joy, B., Steele, G.L.: The Java Language Specification Third Edition. Addison Wesley (2005)
11. MOF: Meta Object Facility, Version 1.4. Object Management Group (2003) formal/2002-04-03.
12. JMI: The Java Metadata Interface(JMI) Specification(Final Release). Java Community Process (2002) JSR-000040.
13. Adaptive, Compuware Corp., Interactive Objects, Sun Mircosystems: MOF 2.0 Facility and Object Lifecycle Specification. Object Management Group (2004) ad/04-04-02, Joint Revised Submission.