A Change Metamodel for the Evolution of MOF-Based Metamodels

Erik Burger[1] and Boris Gruschko[2]

Abstract: The evolution of software systems often produces incompatibilities with existing data and applications. To prevent incompatibilities, changes have to be well-planned, and developers should know the impact of changes on a software system. This consideration also applies to the field of model-driven development, where changes occur with the modification of the underlying metamodels. Models that are instantiated from an earlier metamodel version may not be valid instances of the new version of a metamodel. In contrast to other metamodeling standards like the Eclipse Modeling Framework (EMF), no classification of metamodel changes has been performed yet for the Meta Object Facility (MOF).

The contribution of this paper is the evaluation of the impact of metamodel changes on models. For the formalisation of changes to MOF-based metamodels, a *Change Metamodel* is introduced to describe the transformation of one version of a metamodel to another. The changes are then classifed by their impact on the compatibility to existing model data. The classification is formalised using OCL constraints. The *Change Metamodel* and the change classifications presented in this paper lay the foundation for the implemention of a mechanism that allows metamodel editors to estimate the impact of metamodel changes semi-automatically.

1 Introduction

The divide between the developer and user of model driven tools becomes wider with broader adoption of model driven techniques in software engineering. While providing higher quality tools and increasing the productivity of their application, this divide also introduces the typical development cycles observed in the evolution of any software development tool. The tool developer provides the user with a discrete version of his product, and the user produces content in accordance with the version of the tool available to him. The problem of evolution does not arise if the user applies the same version of the tool throughout a project's lifecycle, or if the user and the tool developer are the same person. In the first case, the non-relevance of the evolution problem is obvious. In the second case the problem does not arise, because the tool user has intimate understanding of tool's inner workings and evolution impact of his doing on the persisted content. In recent years, a number of toolkits for the development of model driven tools have emerged, simplifying the task of tool creation and modification. This development escalates the problem of tool evolution.

[1] Chair for Software Design and Quality, Karlsruhe Institute of Technology, Germany, erik.burger@kit.edu
[2] SAP AG, Walldorf, Germany, boris.gruschko@sap.com

The most visible symptom of the evolution problem in model driven tools is the breaking of model content due to metamodel evolution. This problem arises when the developer of the tool changes the underlying metamodel, rendering the existing content (created by an earlier version of the tool) incompatible to the new metamodel version.

While the problem of metamodel evolution can be handled via active migration, most of the cases could be handled in an automatic manner. Most of the available toolkits for the development of model driven tools have an underlying meta-metamodel to which the metamodels have to adhere. The restrictions imposed by the meta-metamodel can be used to derive a catalogue of all possible metamodel changes and their classification into changes which could lead to the breakage of model content and those which can be introduced in an additive manner. A classification of this kind has been performed for the Ecore metamodel [BGGK07], but is still missing for MOF [MOF05].

In this paper we present an classification scheme of metamodel changes. Furthermore, we present an approach to the attachment of metamodel changes to the actual metamodels (*Change Metamodel*). Another contribution of this paper is the classification of all possible metamodel changes of a MOF 1.4 metamodel according to the proposed classification scheme. This classification also accounts for sequences of metamodel changes and is formalised using OCL constraints.

The work has been performed in scope of the MOIN (MOdeling INfrastructure) project at the SAP AG. MOIN is a MOF 1.4 based repository. [AHK07]

The structure of this paper is as follows: First, an overview of the foundations of metamodel evolution is given. Then, the *Change Metamodel* is introduced and its structure explained. The most interesting cases for the classification of metamodel changes are discussed by example. The complete classification of MOF metamodel change types is presented as an overview table in section 4. Afterwards, the whole process of metamodel evolution description and classification is shown within a common example. The assumptions and limitations of our approach are then mentioned, together with related work before the paper concludes.

2 Foundations

2.1 Difference of Metamodels

The process of metamodel evolution requires to deal with two versions of a metamodel. The difference between two metamodels can be described by a sequence of elementary change operations [AP03]. These operations contain addition and deletion of elements and links, and the modification of element properties. In general, there are two approaches of determining the set of operations describing a change: Either by tracing of single changes or by direct comparison. [Gir06]

In the first approach, tool support is needed in order to record single changes during the editing of the metamodel. The result is a sequence of change operations that reference the original metamodel. Since changes can revert each other, this sequence is not necessarily minimal unless these redundancies are eliminated. Furthermore, if users edit a metamodel through a tool, the operations tool must be matched with the elementary operations mentioned above.

In the second approach, if two metamodels are compared directly, two prerequisites have to be met: Firstly, the underlying infrastructure must support loading two versions of the same metamodel. On the one hand, the matching of elements is simplified by the fact that every element has a unique MOF-ID; on the other hand, this rises the problem that the infrastructure has to deal with different elements that have the same unique identifier. Secondly, an algorithm must be chosen hat calculates a sequence of elementary changes.

2.2 Metamodel Evolution

The process of metamodel evolution can make models inconsistent with the new version of the metamodel. These inconsistencies must be resolved by migrating the models, which is a process that cannot be fully automated. In [GKP07], Gruschko et al. introduce a classification scheme that categorises changes to Ecore-based metamodels into three classes, considering the impact on metamodel instances. We adapt this categorisation for MOF-based metamodels, resulting in the following three *change severities*:

non-breaking < *breaking and resolvable* < *breaking and not resolvable*

- A *non-breaking* change does not require any adaptation of existing models, which is mostly true for additive changes to the metamodel.
- For *breaking and resolvable* changes, an algorithm can be defined to migrate existing instances to the new metamodel version.
- If a *breaking and not resolvable* change occurs, manual interaction is required to make existing models conform to the new metamodel, if possible at all.

When talking about *conformance* of a model to a metamodel, we mean *instantiation conformance* as defined by Steel [SJ04].

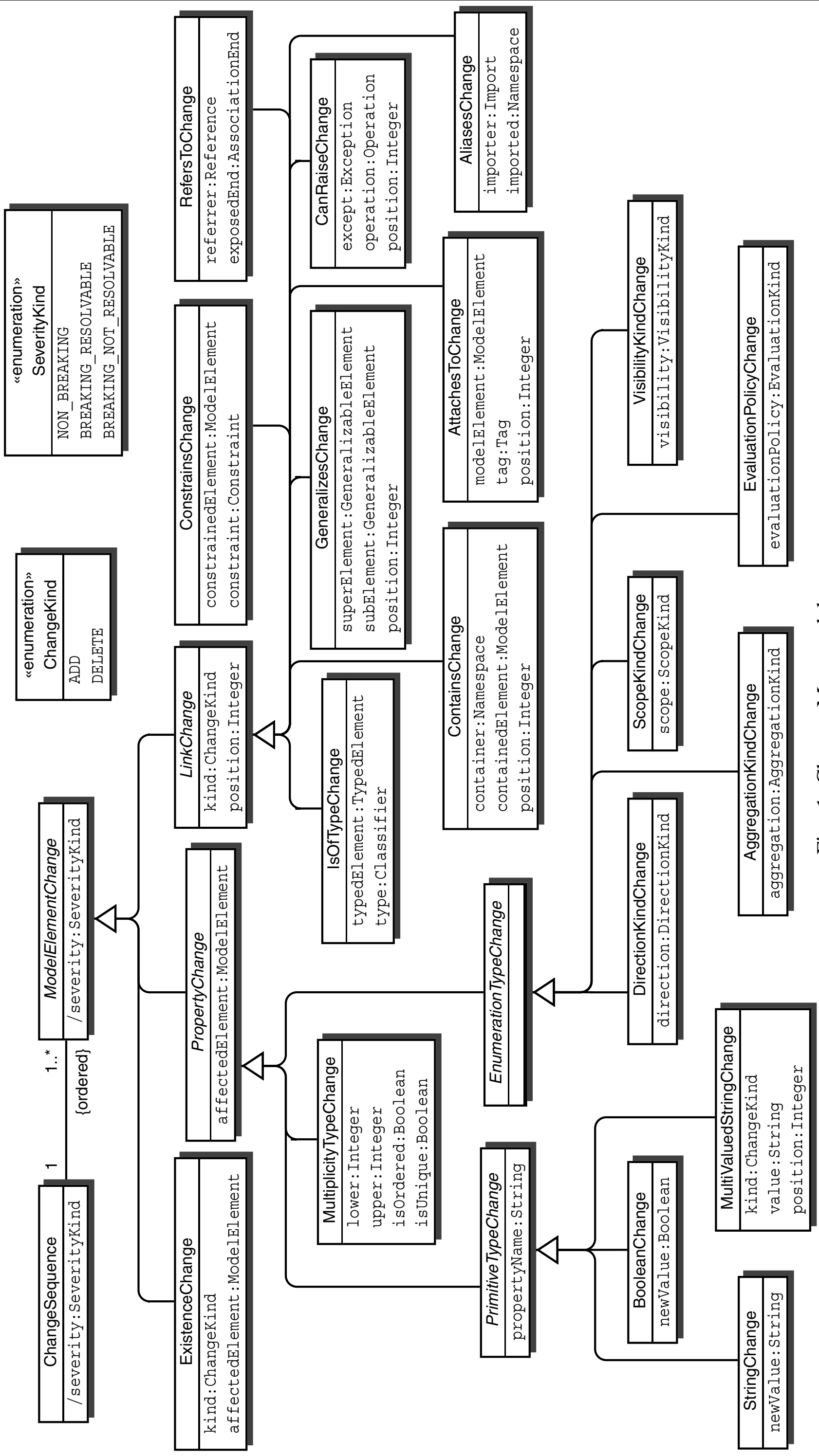

Fig.. 1: Change Metamodel

3 The Change Metamodel

3.1 Definition

For the description of metamodel changes, we introduce a *Change Metamodel* (see figure 1). Instances of this metamodel describe an actual change to a metamodel as a sequence of single change operations, contained in a *ChangeSequence* element. As mentioned in subsection 2.1, all changes in a metamodel can be expressed as a sequence of either additions or deletions of elements and links, or modifications of a property. The *Change Metamodel* is also based on this assumption; the three base classes of the change metamodel cover the addition or deletion of elements *(ExistenceChange)*, the modification of properties *(PropertyChange)* and the deletion or addition of links *(LinkChange)*. These single change types will be described and classified in the following subsections.

3.1.1 ExistenceChange

This change type describes the addition or deletion of an element in the metamodel, i.e. an instance of a MOF class. This covers e.g. classes, attributes, associations etc. In the case of deletion, the *ExistenceChange* instance references an element in the old metamodel; in the case of addition, it references a new element which must be contained in the current Change Metamodel instance.

3.1.2 PropertyChange

A property of a metamodel element represents an attribute in the MOF class of which the element is an instance. This could, for example, be the name of an element or the cardinality of an association end.

Since in the MOF model itself, the attributes of the classes are only typed with *DataType*s (and not with classes), we can specify an actual class in the *Change Metamodel* for every property type. Furthermore, the *propertyName* field is only needed for the *PrimitiveType-Change*, and not for multiplicity or enumeration types, since there is only one attribute of these types in each MOF class.

3.1.3 LinkChange

This class represents all changes in features that are associations in the MOF model. This includes containment, inheritance (generalisation) and typing. Of these, some are not represented as (visible) associations in a model diagram. However, since they are associations in the MOF model, links exist in the metamodels for these concepts, and thus they are covered by this class. For every association that is part of the MOF model, there is a class in the *Change Metamodel*. They contain each two attributes for the ends of the association, which are named after the roles in the MOF model. Changing a link is always expressed as either a deletion or addition. This approach is common practise in other fields, e.g. database schema migration [Mon93, p. 42].

For associations that are ordered, the field *position* describes the position of a newly added element. For deletions, this value is ignored.

3.2 Classification

The classification of a change is expressed by the derived attribute *severity* in the respective classes of the *Change Metamodel*. The semantics of this attribute are formally described by OCL constraints that cover all the cases shown in the table in section 4. A complete listing of all constraints can be found in [Bur08].

3.2.1 Overall Severity

Since changes are contained in change sequences, not only the severity of a single change has to be regarded, but also the overall severity of a sequence of changes. Intuitively, the overall severity can be determined by finding the maximum (according to the order presented in subsection 2.2) of the severities of all singular changes in the change sequence.

However, if the change severity of a single change is only based on the effect of that single change, the overall severity is only equal to the maximum of the sequence's severities in trivial cases. If a sequence of changes is applied to a metamodel, it is possible that a combination of changes has a lower severity than any of the single changes. A simple example is the consequent addition and deletion of the same element; the latter change reverts the first one, and there is no effect on the metamodel at all.

As a consequence of this, the OCL constraints that describe the severity of a *ModelElementChange* also account for other changes in the same *ChangeSequence*. The constraints express the severity of a single change in the context of the complete change. If a change sequence only contains one element, the constraint expresses the severity for a singular change. Thus, the maximum severity yields the correct result for the overall severity of a sequence of changes.

3.2.2 Interesting Cases by Example

In this section, we will show two of the more difficult cases, where the determination of change severities is non-trivial.

Deletion of a Class (cf. Figure 2) If classes are deleted in the metamodel, the corresponding M1 instances must also be deleted. This change is normally resolvable, but can be *breaking and not resolvable* if a situation similar to the one shown in Figure 2 occurs: The instance of *Type2* has a link to an instance of *Subtype1*. If *Subtype1* is deleted, the association still exists in the metamodel, since it is connected to the superclass *Type1* of the deleted class. As a consequence of the deletion of *Subtype1*, all its instances and the links to these instances are also deleted, leaving only the instance of *Type2*. Now the model data is invalid, since there is no link to an instance of *Type1*, which is required by the minimum cardinality of *1* in the metamodel.

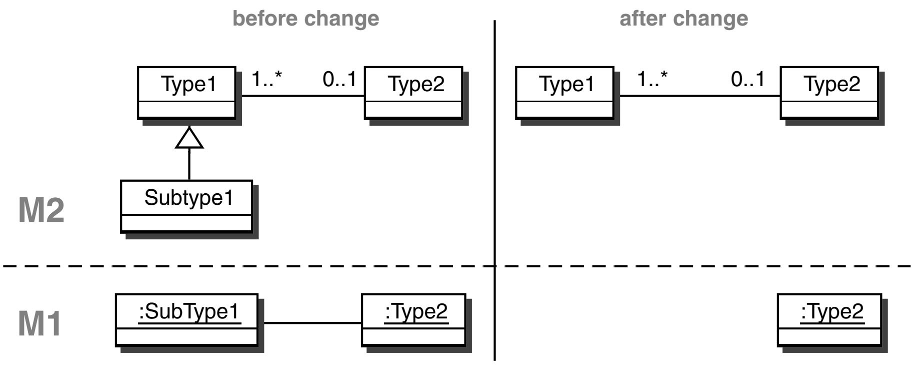

Fig.. 2: Example: Deletion of a class makes M1 data invalid

Association end moved to superclass (cf. Figure 3) If one end of an association is moved to another type, the change consists of a type change of the affected *AssociationEnd*. If the situation is similar to Figure 3, existing M1 data will become invalid. In the example, *EndB* is changed to a supertype. In the M1 data, the instances of *Type1* and *SubType2* will still be valid after the change, since the association still exists. Only the instance of *Type2* is invalid since there is no association to an instance of *Type1* despite the minimum cardinality *1* of *EndA*. This makes the change *breaking and not resolvable*.

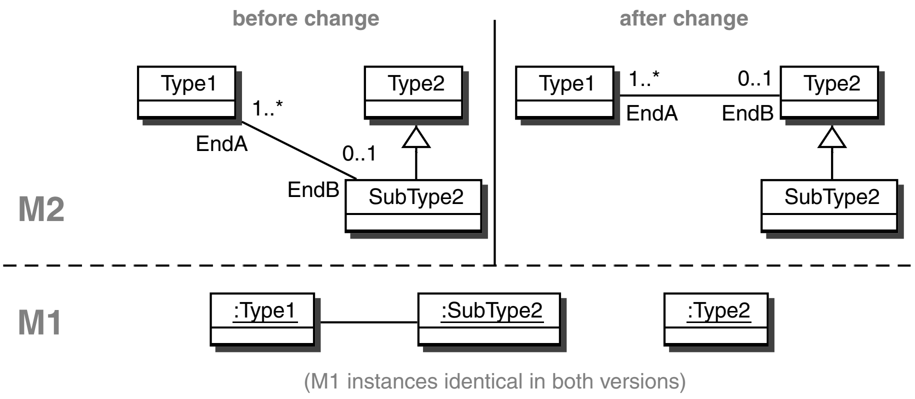

Fig.. 3: Example: Type change on an association end

4 Classification of MOF Metamodel Changes

The table shows the change severities of the single cases, grouped by the classes of the *Change Metamodel*, which can take the values *non-breaking* (nb), *breaking and resolvable* (br) and *breaking and not resolvable* (bn). For a comprehensive explanation, examples and the OCL constraints of all special cases, please refer to [Bur08].

MOF change type		nb	br	bn	condition
ExistenceChange					
`Class`	add	✓			additive
	delete			✓	supertype has mandatory association end
			✓		other cases
`Attribute`	add			✓	minimum cardinality >0, no initializer
			✓		minimum cardinality >0, initializer exists
		✓			minimum cardinality $=0$
	delete		✓		
`Association`	add			✓	minimum cardinality >0
		✓			minimum cardinality >0, no instances
		✓			minimum cardinality $=0$
	delete		✓		
`AssociationEnd`					same as for `Association`
`Reference`	add	✓			
	delete	✓			
`Package`	add	✓			
	delete		✓		contents resolvable
				✓	contents not resolvable
`Import`	add	✓			
	delete		✓		elements resolvable otherwise
				✓	elements not resolvable
`Tag`	add	✓			
	delete	✓			
`Constraint`	add			✓	
	delete	✓			
`Operation`	add/delete	✓			
`Exception`	add/delete	✓			
`Parameter`	add/delete	✓			
`DataType`	add/delete	✓			
`StructureField`	add/delete	✓			

PropertyChange

`ModelElement`	name		✓		refactoring, unique ids
	annotation	✓			
`Generalizable Element`	isRoot	✓			
	isLeaf	✓			
	isAbstract	✓			`true` → `false`
			✓		`false` → `true`
	visibility	✓			increase
				✓	decrease
`Association`	isDerived	✓			`true` → `false`
				✓	`false` → `true`
`Class`	isSingleton	✓			`true` → `false`
				✓	`false` → `true`
`Constraint`	expression			✓	
	language			✓	
	evaluationPolicy	✓			
`Feature`	scope			✓	`instance_level` → `classifier_level`
		✓			`classifier_level` → `instance_level`
	visibility	✓			increase
				✓	decrease
`StructuralFeature`	multiplicity	✓			widening
				✓	narrowing
	isChangeable	✓			
`Attribute`	isDerived	✓			`true` → `false`
				✓	`false` → `true`
`Operation`	isQuery	✓			
`Constant`	value	✓			
`Parameter`	direction	✓			
	multiplicity	✓			
`AssociationEnd`	isNavigable	✓			
	aggregation	✓			`composite` → `none`
				✓	`none` → `composite`, composition closure rule violated
		✓			closure rule not violated
	multiplicity	✓			widening
				✓	narrowing
	isChangeable	✓			
`EnumerationType`	labels	✓			addition
				✓	delete/modify

LinkChange

Contains		✓			not mandatory; to supertype
			✓		not mandatory; to other type
		✓			mandatory; to new or abstract supertype
				✓	mandatory; to other type
Generalizes	add			✓	supertype contains mandatory features or associations
			✓		supertype contains non-mandatory features or associations
		✓			supertype contains no features or associations
	delete		✓		no associations to supertype or type compatible; new features in supertype
		✓			no associations to supertype or type compatible; no new features in supertype
				✓	supertype has adjacent mandatory association ends; change not type compatible
			✓		supertype has adjacent non-mandatory association ends; change not type compatible
IsOfType		✓			new type is supertype; general case
				✓	new type is supertype; special case for association end with mandatory other end
		✓			new type is subtype and instances are type compatible
				✓	new type is no sub- or supertype
RefersTo		✓			
CanRaise		✓			

5 Example

The example seen in Figure 4 shows a common case of metamodel evolution. For a given type (*Type1*), a supertype is introduced (*Type2*). The attribute *attr1* is a mandatory feature, since it has a lower cardinality of 1. It is moved to the new supertype.

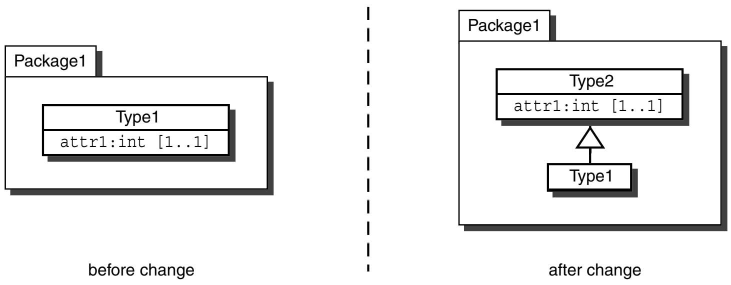

Fig.. 4: Example metamodel

5.1 Change Description

The transition from Version 1 to Version 2 is described by instances of the *Change Metamodel*, as depicted in Figure 5. First, the new type is created in *Change1*. Then, the generalization between the types is added (*Change2*). The move operation for the attribute *attr1* is described as the deletion and addition of the *Contains* link between the attribute and the types in *Change3* and *Change4*.

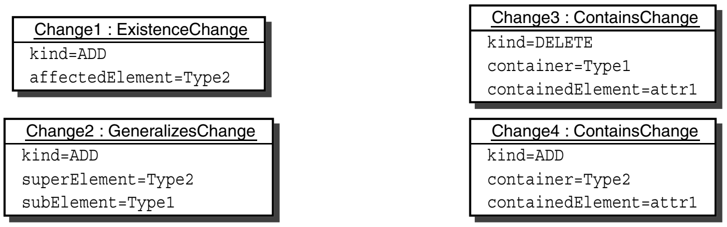

Fig.. 5: Change steps as Change Metamodel instances

5.2 Classification

Since the first change is purely additive, it is *non-breaking*. In the second change, a generalisation link is added. If we looked at the change steps until here, *Type2* would not contain any features, so one could assume that the change were trivially *non-breaking*. However, in the OCL constraints, all other changes of a sequence are also taken into account, as stated in subsubsection 3.2.1. This also includes the changes which are performed after the current change. In the resulting metamodel, *Type2* will contain a mandatory attribute, which would make the change *breaking and not resolvable* because there are no values for it in the instances of *Type1*. But since all of the subtypes of *Type2* (in this case, only *Type1*) contained an attribute of the same name and of the same type in the former metamodel version, existing valid M1 instances contain values for this attribute. This is why the addition of the generalisation is *non-breaking* in this case.

Finally, the mandatory association is moved to the new supertype. This way, all instances of *Type1* are still valid. For *Type2*, the situation is more complex, since a mandatory feature was introduced, which could cause instances to become invalid, as they do not have a value for this feature. But since *Type2* was newly created in the course of this change sequence, there cannot be any instances in existing M1 data, and so the change is also *non-breaking*.

For the overall change, this means that the change is *non-breaking*. The interesting thing about this example is the fact that the single changes would have different severities if they were analysed as singular changes without the context of the change sequence. In total, this would yield a wrong estimation of the overall severity. Only if all four change steps are regarded, the severities can be determined correctly.

5.3 Example OCL constraint

In this section, we take a look at the OCL constraint for *Change3*. As mentioned in the paragraph above, the deletion of the containment link between *attr1* and *Type1* is *non-breaking* because the attribute is moved to a supertype. This is expressed in lines 6–13 of the following constraint.

```
 1  context ContainsChange
 2  inv: (self.container.oclIsTypeOf(Class) and
 3     self.containedElement.oclIsKindOf(StructuralFeature))
 4  implies
 5  if (self.kind=DELETE) then
 6     if self.changeSequence.changes -> exists(c|
 7        c.oclIsTypeOf(ContainsChange) and c.kind = ADD and
 8        (c.containedElement = self.containedElement or
 9        c.containedElement.similar(self.containedElement)) and
10        self.newSupertypesExtended(self.container)
11           -> contains(c.container))
12        -- feature moved to supertype
13        then self.severity = NON_BREAKING
14        -- feature deleted or moved elsewhere
15        else self.severity = BREAKING_RESOLVABLE
16     endif
17     else -- (self.kind=ADD)
18     -- (omitted here)
19  endif
```

Listing 1: Severity Constraint for Change3 (excerpt)

Here, the interesting part is the function *newSupertypesExtended* (line 11), which is the transitive closure of the *newSupertype* helper function shown below. The function calculates the supertype structure in the new version of the metamodel. Functions of this type are necessary since we cannot reference elements in the new version of the metamodel.

```
context ModelElementChange::newSupertypes(GeneralizableElement
                        element) : Set(GeneralizableElement)
post: result = element.supertypes
-> including(this.changeSequence.changes -> select(ch|
   ch.oclIsTypeOf(GeneralizesChange) and
   ch.subtype = element and ch.kind = ADD) -> collect(supertype)
   )
-> excluding(this.changeSequence.changes->select(ch|
   ch.oclIsTypeOf(GeneralizesChange) and
   ch.subtype = element and
   ch.kind = DELETE) -> collect(supertype)
   )
```

Listing 2: Helper function

6 Assumptions/Limitations

6.1 Unique Identifiers

As seen in the example above, changes in containment or generalisation are expressed as two operations (delete and add) in the *Change Metamodel*. This makes it more difficult to recognize the semantics of such a change. However, the description is unambiguous since in MOF 1.4, every element has a unique identifier [MOF05].

In order to detect a change in containment, the change sequence has to be searched for elements of the same type (e.g. *ContainsChange*) with different change kind (delete, add) and the identical affected element, which can be determined by the element's unique identifier. The decision to describe changes in this way is backed up by the fact that containment, typing and other relations are expressed by instances of MOF associations like *Contains* and *IsOfType*, i.e. links in a MOF-based metamodel. Since links do not have element identity, it is not possible to distinguish the "modification" of a single link from its deletion and the creation of a new link that connects to one of the former link's ends. Only the MOF constraints which state that an element must be contained in a container or that a typed element must have a type ensure that for every deleted link of these types, a new one is created. Otherwise, the metamodel would not be a valid MOF instance. Since we did not want to hard-code these contraints in the structure of our metamodel, all modifications to associations in MOF are mapped to subtypes of *LinkChange*.

6.2 References to Metamodels

A *Change Metamodel* instance only references elements in one version of the metamodel. The *Change Metamodel* instance is applied like a patch and generates the new version. If new elements are added during the progress of modification, they must be contained with the *Change Metamodel* instance.

This method has the advantage of not having to deal with two versions of the same element, which would be the case if the change description referenced elements from both evolutionary stages. A *Change Metamodel* instance has an inverse that references only elements of the newer metamodel version.

6.3 M1-agnostic analysis

The severity of a *Change Metamodel* instance depends on its own structure and the structure of the metamodel it references, i.e. the M2 level. In general, the nature of M1 instances also has an influence on the severity of changes to a M2 model. For example, if a class does not have instances, all changes to this class would be non-breaking.

But since metamodels and M1 instances are not necessarily used by the same person, even in the same environment, it may be impossible for the editor of a metamodel to know about all or any M1 instances. For this reason, the classification of this paper is a worst-case assumption: All severities in section 4 have been calculated for the worst-case of possible M1 instances.

7 Related Work

The problem of finding a minimal edit script for hierarchically structured information is adressed in [CRGMW96]. The algorithm presented there calculates a minimal edit script for different version of a graph, under the assumption that no unique identifiers are present. The edit script is composed of atomic editing operations on graphs. For the process of finding a delta of two metamodels without the support of traces, this algorithm could be used to determine a sequence of atomic change steps.

This algorithm is also used by Girschick in [Gir06] to compare UML class diagrams. Girschick introduces the UMLDiff$_{cld}$ algorithm that is based on elementary transformation options. The purpose of this algorithm lies mainly in difference visualisation for graphical UML tools. The visualisation could be generalised to be used with any kind of MOF models.

A metamodel for the description of model deltas has been presented by David Hearnden in the *Deltaware* project [Hea07, pp. 72]. The *Delta* model is based on MOF 2.0 and allows the description of model deltas with the help of identity maps. While the addition and deletion of elements and changes of properties are described thoroughly, the *Delta* model lacks descriptive methods for changes in the model structure, like containment or generalisation. The focus of this work lies on metamodel transformations, for which evolutionary aspects are formalised using the Tefkat language.

Ciccheti et al. [CRP07] propose a metamodel-independent approach for the description of model deltas that is also based on atomic change operations. They also describe model deltas through instances of a difference metamodel, which is derived from a concrete model delta. In contrast, the change metamodel in this paper is fixed, since the meta-model is always the MOF model, and so the semantics of changes can be described more specifically, regarding the effects of metamodel changes to existing instances.

The migration of model instances under metamodel evolution is denoted as adaptation and co-adaptation by Wachsmuth in [Wac07]. The steps of adaptation are described with QVT Relations with respect to instance preservation on the model side. For this purpose, a set of metamodel relations is defined, which allow the estimation of the impact of a metamodel adaption, based on the layout of model data.

For EMF, there are several approaches for the description of metamodel evolution. Becker et al. suggest a process model for semi-automatic evolution, including the change classification that this paper is based on [BGGK07]. The idea behind this process model is that a metamodeling infrastructure should assist the user with the transformation of existing model data by distinguishing automatically adaptable changes from those that need manual interaction.

Herrmannsdörfer defines a language for metamodel evolution and model tranformation called *COPE* [HBJ08], which is used for the evolution of Ecore models. The approach includes *coupled trans-*

actions for the M2 and M1 level which are reusable, as well as tool support. Coupled evolution scenarios are also described by Vermolen et al. in [VV08]. While these approaches focus on the transformation of existing model data, the change classification of this paper is primarily directed at the analysis of the impact of changes to the metamodel.

8 Conclusion

The change metamodel presented in this paper allows a description of the metamodel evolution process of MOF-based metamodels using MOF itself as a description language. Based on this, the severity of changes to a metamodel can be determined for single changes, and, more importantly, for complex change sequences using the change severity classification of section 4. This classification can be used to make statements about the compatibility between arbitrary existing model instances and new versions of the metamodel. The severity classification of changes expressed in the *Change Metamodel* has been noted formally using OCL, so that the results of an automatic or manual classification can be checked by the respective modeling infrastructure. This makes it possible to use the results of the change classifications presented here in a platform-independent way. For manual evaluation of change severities, an overview table has been created.

In a metamodeling infrastructure which allows the comparison of different versions of the same metamodel, the classification presented in this paper could be used to automatically determine the impact of a change to a certain metamodel on existing model data. This would require an automatic calculation of a change metamodel instance from either two versions of a metamodel or from the current editing process of a persisted metamodel. The model editing tool could then determine the severity of the changes, which would enable the metamodel editor to decide about changes to the metamodel regarding the projected impact on existing data as well as interface compatibility of the generated software. The implementation such guidance tools for model editing tools remains subject to future work. For users of the modeling tools, the formal description of metamodel evolution changes would ease the migration of their existing model data to new versions of that software.

The incorporation of techniques alleviating metamodel evolution into modeling infrastrcutures would allow for faster adoptation of modeling tools to user needs.

References

[AHK07] Michael Altenhofen, Thomas Hettel, and Stefan Kusterer. OCL Support in an Industrial Environment. In *Models in Software Engineering. Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, volume 4364 of *Lecture Notes in Computer Science*, pages 169–178, Berlin/Heidelberg, 2007. Springer Verlag.

[AP03] Marcus Alanen and Ivan Porres. Difference and Union of Models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *"UML 2003" – The Unified Modeling Language, Modeling Languages and Applications 6th International Conference, San Francisco, CA, USA, October 20–24, 2003, Proceedings*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17, Berlin/Heidelberg, 2003. Springer Verlag.

[BGGK07] Steffen Becker, Thomas Goldschmidt, Boris Gruschko, and Heiko Koziolek. A Process Model and Classification Scheme for Semi-Automatic Meta-Model Evolution. In *Proc. 1st Workshop "MDD, SOA und IT-Management" (MSI'07)*, pages 35–46. GiTO-Verlag, April 2007.

[Bur08] Erik Burger. Metamodel Evolution in the Context of a MOF-Based Metamodeling Infrastructure. Master's thesis, Universität Karlsruhe (TH), September 2008. `http://sdqweb.ipd.kit.edu/publications/pdfs/burger2008a.pdf`.

[CRGMW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change Detection in Hierarchically Structured Information. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 493–504. ACM Press, 1996.

[CRP07] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, 2007. `http://www.jot.fm/issues/issue_2007_10/paper9/index.html`.

[Gir06] Martin Girschick. Difference Detection and Visualization in UML Class Diagrams. Technical Report TUD-CS-2006-5, Technische Universität Darmstadt, 2006.

[GKP07] Boris Gruschko, Dimitrios S. Kolovos, and Richard F. Paige. Towards Synchronizing Models with Evolving Metamodels. In *In Proc. Int. Workshop on Model-Driven Software Evolution held with the ECSMR*, 2007.

[HBJ08] Markus Herrmannsdörfer, Sebastian Benz, and Elmar Jürgens. COPE: A Language for the Coupled Evolution of Metamodels and Models. In *1st International Workshop on Model Co-Evolution and Consistency Management*, 2008. `http://www.info.fundp.ac.be/mccm/2008/wp-content/uploads/2008/09/9-herrmannsdoerfer.pdf`.

[Hea07] David Hearnden. *Deltaware: Incremental Change Propagation for Automating Software Evolution in the Model-Driven Architecture*. PhD thesis, University of Queensland, School of ITEE, October 2007.

[MOF05] Object Management Group. *Meta Object Facility (MOF) Specification, Version 1.4*, July 2005. `http://www.omg.org/docs/formal/05-05-05.pdf`.

[Mon93] Simon Monk. *A Model for Schema Evolution in Object-Oriented Database Systems*. PhD thesis, Lancaster University, February 1993.

[SJ04] Jim Steel and Jean-Marc Jézéquel. Typing Relationships in MDA. Technical Report 17, University of Kent at Canterbury Computing Laboratory, 2004.

[VV08] Sander Vermolen and Eelco Visser. Heterogeneous Coupled Evolution of Software Languages. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulose, France, September 28 - October 3*, volume 5301 of *Lecture Notes in Computer Science*, pages 630–644, Berlin/Heidelberg, 2008. Springer Verlag.

[Wac07] Guido Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In *ECOOP'07 – Object-oriented programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624, Berlin/Heidelberg, 2007. Springer Verlag.