A Formalisation of Analysis-based Model Migration

Ingrid Chieh Yu and Henning Berg
Department of Informatics, University of Oslo, Oslo, Norway

Keywords: Metamodelling, Co-evolution, Adaptation, Composition, Variability.

Abstract: Supporting adaptation of metamodels is essential for realising Model-Driven Engineering. However, adapting and changing metamodels impact other artefacts of the metamodelling ecosystem. In particular, conformant models will no longer be valid instances of their changed metamodel. This gives rise to co-evolution issues where metamodels and models are no longer synchronised. This is critical as systems become inconsistent. A typical approach for re-establishing conformance is to manually craft transformations which update existing models for the new metamodel variant. In this paper we present an analysis-based approach that addresses this concern. The approach enables an arbitrary number of metamodels to evolve based on an adaptation strategy. During analysis we accumulate information required to automatically transform existing models to ensure conformance. We formalise the approach and prove model conformance.

1 INTRODUCTION

From a broad view, software engineering is the process of understanding a system and representing it as a set of models that can be understood by computers. The models allow reasoning about the system and perform various analyses and evaluation of system-specific data; whether the system already exists or is to be built. Modelling is the process of creating abstractions of a system. A system may itself be a model (Favre, 2004)(Seidewitz, 2003), something that has given us the notion of *metamodelling*. Metamodelling is the process of formalising models by indentifying their structure and semantics. The result of such a process is one or more metamodels.

Metamodelling is a key process in several *Model-Driven Engineering (MDE)* (Kent, 2002) disciplines including language design, product line engineering, variability management and domain modelling. The promised benefits of using model-oriented approaches are increased efficiency, quality and consistency in software. Using models also supports code generation, multi-view modelling and improved verification of system properties. There are clear incentives in the industry for applying MDE approaches, as shorter time to market for software products and mechanisms that support variability are needed.

A metamodel is defined according to the rules and constraints of a metamodelling architecture or metamodelling framework. The prominent metamodelling architecture in the industry is the *Object Management Group (OMG)* standardised *MetaObject Facility (MOF)* (ObjectManagementGroup, 2014), and a variant based on *Essential MOF (EMOF)* implemented in the *Eclipse Modeling Framework (EMF)* named *Ecore* (TheEclipseFoundation, 2014). Both MOF and Ecore have concepts for creating standalone metamodels of arbitrary domains. However, the architectures do not have built-in mechanisms that address co-evolution of models as metamodels change. One such change or adaptation is composition of metamodels. Composition may be used to increase a metamodel's expressiveness or to weave in variability (Didonet Del Fabro et al., 2006)(Fleurey et al., 2008)(Kolovos et al., 2006a)(Morin et al., 2008)(Morin et al., 2009).

A model is said to conform to its metamodel when all elements in the model are legal instances of structural elements found in the metamodel. Changing the metamodel typically causes the conformance relation to be broken. Consequently, the models are no longer valid according to the changed metamodel. This is unfortunate as it creates system inconsistencies.

There are three main categories of approaches available that address how models, transformations and tools may adapt to changes of evolving metamodels (Rose et al., 2009). These are *manual specification*, *operator-based co-evolution* and *metamodel matching*, where the latter category may be further divided into *change recording* and *differencing*. In this paper we present a novel approach of model migra-

Yu I. and Berg H..
A Formalisation of Analysis-based Model Migration.
DOI: 10.5220/0005240900860098
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 86-98
ISBN: 978-989-758-083-3

tion by using analysis as a core process. Specifically, our approach is a hybrid between operator-based co-evolution and metamodel matching, where information required to re-establish model conformance is generated automatically during the analysis step. All model migration approaches we have come across are limited to what we refer to as *serial evolution*, i.e. model migration is only addressed for evolution of one metamodel at a time, where the metamodel is evolved from an initial version to a revised version independently of other metamodels. However, this is an ideal case. In practice, metamodels may evolve in parallel due to dependencies that exist or are introduced *between* the metamodels. We refer to this as *parallel evolution*. Examples of dependencies that may be introduced are when metamodels are composed, e.g. by merging or interfacing classes. In the general case, there may be an arbitrary number of metamodels for which dependencies are introduced. Hence, evolving one metamodel should be reflected on the other metamodels as well. Furthermore, the evolution of the metamodels should be propagated to all existing models of all the metamodels. That is, existing models conforming to the metamodels must *all* be updated to conform to the new metamodel versions or resulting composite metamodel. This is not trivial due to the dependencies introduced, and is the reason why analysis is required. The proposed analysis framework is able to process an arbitrary number of metamodels simultaneously, and accumulates information about changes performed on the metamodels. The information is later used as input to model-to-model transformations that update existing models.

Changing or adapting metamodels do not only impact the models. Changing a metamodel also severely impacts other artefacts in the metamodelling ecosystem that are defined relative to the metamodel. Such artefacts include editors, concrete syntaxes, transformations and interpreters. Changing or altering a metamodel results in co-evolution issues with these artefacts, as the metamodel and artefacts do not evolve in a synchronised manner. Our approach is also applicable for co-evolution of artefacts. Specifically, the information generated indicates what changes that must be performed in the artefacts for these to be valid with respect to evolved metamodels. In this paper we limit our scope to describing how the conformance between models and metamodels may be re-established as the metamodels are changed or adapted. We exemplify the approach by studying how two metamodels are composed (we view metamodel composition as an adaptation).

We formalise an analysis framework based on a set of *basic adaptation operations*. The adaptation operations are fine grained and can be traced back to the catalogue of operators presented in (Herrmannsdoerfer et al., 2011). The analysis framework systematically generates information which is needed for re-establishing conformance between the adapted metamodels and existing models. We focus on model structure and leave semantics for future work. We do not consider matching of metamodels using heuristics (Del Fabro and Valduriez, 2007), as this is outside of our scope. However, we give our thoughts on this topic with respect to our approch in Sec. 6 where we briefly describe how to incorporate the use of ontologies in the analysis.

The contribution of this paper is two-fold. First, we discuss a new approach for model migration and how conformance between models and metamodels can be re-established as metamodels are adapted or composed. The approach is applicable to other artefacts in the metamodelling ecosystem, e.g. it supports metamodel-tool co-evolution. Second, we present a thorough formalisation of the analysis that is used in our approach. The formalisation not only describes our approach in details, but it gives a formalisation of how model conformance may be re-established as metamodels evolve, i.e. metamodel-model co-evolution. There is not much work available that formalises metamodel-model co-evolution. As far as we know, the only formalisations on this topic is in the form of typed graphs and category theory (Taentzer et al., 2012)(Mantz et al., 2010).

Paper overview. Sec. 2 gives a motivating example, Sec. 3 introduces adaptation operations for metamodel evolution, Sec. 4 addresses model conformance, Sec. 5 discusses related work, and Sec. 6 concludes the paper.

2 MOTIVATING EXAMPLE

We will motivate and explain our approach using a simplified example in the domain of ecommerce. The example will be developed throughout the paper. Ecommerce is an industry concerned with sale and purchase of products and services using Internet technologies. Ecommerce is a multi-facetted concept comprising e.g. financing and payment, pricing and marketing analysis, logistics, product modelling, business modelling and business strategies. All of these concerns are modelled in some way - either informal or in a more formal manner using e.g. a set of UML models. An increasingly popular approach is to create a set of domain-specific languages (DSLs) for each of the major concerns. This approach allows stakeholders to model their solution directly at the

level of ecommerce domain concepts. That is, each domain-specific language contains a set of constructs for modelling of one explicit concern. An ecommerce solution is then the sum of all the models that describe it.

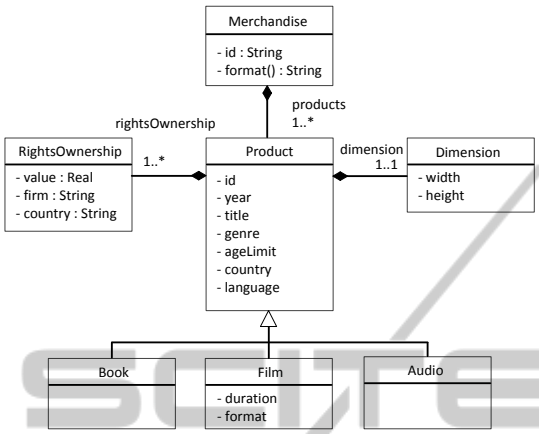

Figure 1: Metamodel for modelling of products/merchandise.

To keep our example manageable, we will only focus on two aspects of an ecommerce solution: product modelling and business context modelling. Product modelling, as we use the term here, is the process of describing a product intended for sale - including all its metadata. Business context modelling deals with concerns such as ownership, regions where a given product or group of products should be available, price and such.

We assume that the conceptual domain of our example is sale of media products like books, films and audio. That is, our target software solution is that of an online retailer like Amazon. A product, see Figure 1, has attributes like year of publication or release, title, genre, etc. A product has a dimension and a set of copyright owners (note that most attribute types and attribute multiplicites are excluded for clarity reasons). A product model describes either one specific product or a product group. The product model does not describe any aspects related to sale of this product.

The trading perspective is covered by the business context model, see Figure 2, which describes the business dimension of a product or product group. That is, the business model puts a product in a business context. This includes describing the price of the product, availability, fees and eventual discounts. A product may have several business contexts. As an example, a product may have different prices and discounts depending on the region of sale.

The two metamodels, supporting modelling of concerns in two different domains, may be combined

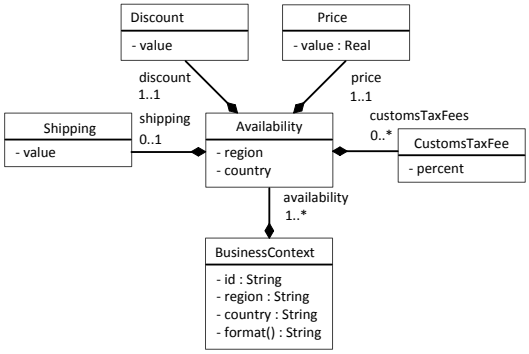

Figure 2: Metamodel for modelling of business contexts.

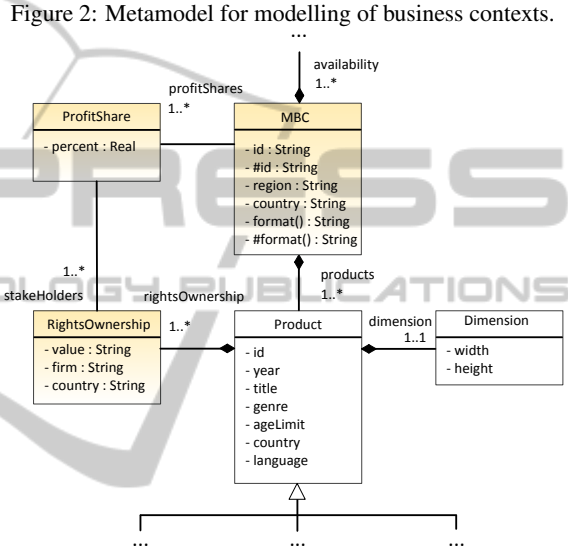

Figure 3: The metamodel resulting from composing the metamodels for product and business context modelling.

to form a new metamodel for ecommerce, see Figure 3. The new metamodel has a class named MBC that is a combination of the Merchandise and BusinessContext classes. The metamodel is also extended with a new ProfitShare concept. In the subsequent sections, we will demonstrate how the new metamodel is systematically derived.

3 METAMODEL ADAPTATIONS

We base our analysis framework and example on an excerpt of the MOF structural concepts (ObjectManagementGroup, 2014)(Steel and Jzquel, 2007). The excerpt represents the most essential concepts of MOF:

a

Metamodel	::=	package P $\{\overline{ClassDecl}\}$
ClassDecl	::=	class C extends $\overline{C'}$ $\{\overline{PropDecl}\ \overline{OpDecl}\}$
PropDecl	::=	t P $(Multi)?$ $(containment)?$ $(\#\ P')?$
Multi	::=	$[lower\ ..\ upper]$
OpDecl	::=	t $O(\overline{t_i\ P_i})$
$t \in Type$	::=	*Primitive* | C | $set\langle C\rangle$ | Void
Primitive	::=	Boolean | Integer | Real | String

A metamodel consists of a package with one or more classes. (In general a metamodel may have several packages, which can be easily accommodated by our approach.) A class may have an arbitrary number of properties and operations. Properties do either have primitive types or class types. Hence, they either act as attributes or references. *containment* takes the values true or false, whereas *lower* and *upper* are natural numbers. In this paper, we use the term *well-typed metamodel* as a model satisfying the requirements in (ObjectManagementGroup, 2014). Specifically we have the following typing restrictions on property and operation definitions in MOF metamodels that inherently affect the proposed adaptation rules:

- A property name can only occur once in a class hierarchy
- Operations may be overloaded in the same class, but only occur once with the same signature
- Overloaded operations in a class hierarchy that have the same parameter types require the same return type

We formalise metamodel adaptations in terms of *basic adaptation operations*. We consider constructive operations that are not neccessarily model-preserving (Herrmannsdoerfer et al., 2011). The operations can be combined to form more complex adaptation strategies. We do not aim at theoretical completeness but rather focus on common practical adaptations. The framework is structured in such a way that it can easily be extended by adding more operations.

Metamodels can be adapted using the following basic adaptation operations, where N and N' are class names, C a class term, P a property definition, O an operation definition, and $\mathcal{R}$ a tuple $\langle N, \overline{P}\rangle$:

- $merge(N, N')$ creates a new class by merging two classes named N and N'
- $addClass(C)$ adds a new class definition
- $addSuperClass(N, N')$ extends the class named N with a new superclass N'
- $addInterfaceClass(C, \mathcal{R})$ adds a class C that serves to bridge classes according to the relations specified in $\mathcal{R}$
- $override(N, N')$ overrides the class named N with the definition of N'
- $addProp(N, P)$ extends class N with a new property P
- $addOp(N, O)$ extends a class N with a new operation O
- $addBiProp(N, P, N', P')$ extends class N with a property P and N' with a property P', where P and P' combined describe a bi-directional relation

Note that the *merge*, *addSuperClass* and *override* operations differ from other operations in the sense that existing models may have objects that relate to the structure of the operands (i.e. the classes). That is, *addClass*, *addInterfaceClass*, *addProp*, *addOp* and *addBiProp* deal with adding new metaelements for which no existing objects relate to. Still, it may be required to generate default objects of added structure. We will return to this later.

A sequence of adaptation operations, $\varphi_1 \cdot \varphi_2 \cdot \cdots \cdot \varphi_n$, defines an *adaptation strategy* Φ for a particular metamodel adaptation. An adaptation strategy can be seen as a description that differentiates one metamodel variant from another. The adaptation strategy and sequence of operations are given by the user, e.g. as provided using a graphical tool (i.e. not specified textually). Figure 4 gives a conceptual overview of the analysis framework and how two metamodels are composed by using the *merge* operation. Specifically, the classes represented by filled boxes are merged according to a merge strategy specified by the user. Notice the references to the classes marked with x. The multiplicities have a lower bound equal to 1, which indicates that objects of the classes need to be created to have conformant models. We will later see how this is addressed by the framework.

3.1 The Analysis Environment

The analysis environment for metamodel adaptations consists of a tuple $\langle \mathcal{E}, \sigma, \delta\rangle$:

Definition 1. *The environment mapping $\mathcal{E}$ maps class names Nc to class definitions: $\mathcal{E} : Nc \to Class$, where Class is a set of classes $\langle Nc, Inh, Prop, Op\rangle$. Nc is the name of the class. Inh is a list of class names defining class inheritance (direct superclasses). Prop is a set of properties $\langle Type, Np, Multi, Cont, Opp\rangle$ where Type is a type, Np a property name, Multi the property's multiplicity comprising a lower and upper bound, Cont describing whether the property is of containment type, and Opp the name of an optional opposite relation. Op is a set of class operations $\langle Type, No, Param\rangle$ where Type is the operation's return type, No is the operation's name and Param is a list of input parameter declarations.*

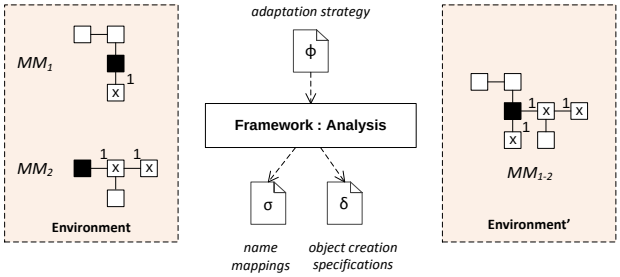

Figure 4: Illustration of the analysis and the *merge* operation.

Dot notation is used to access the elements of tuples such as properties and operations; e.g. $(C,\overline{I},\overline{P},\overline{O}).Prop = \overline{P}$, where we use overlines to denote sets or list structures. $Prop(N)$ denotes a subset of properties in $Prop$ with name N and $Prop.Np$ gives all property names in $Prop$. Similar notation is used for Op . The empty list is denoted ε.

Definition 2. *The environment mapping* σ *consists of a family of mappings* $\langle \sigma_c, \sigma_p, \sigma_o \rangle$*:*

$$\begin{array}{lcl} \sigma_c & : & Nc \rightarrow Nc \\ \sigma_p & : & Np \overset{Nc}{\rightarrow} Np \\ \sigma_o & : & No \overset{Nc}{\rightarrow} No \end{array}$$

where Nc *,* Np *and* No *are class, property and operation names, respectively. For* σ_p *and* σ_o *,* Nc *(above the arrow) specifies the containing class of the property and operation.*

The family of mappings contains substitutions accumulated during the analysis of classes, properties and operations, respectively. Substitutions are introduced in the adaptation analysis to resolve conflicts associated with overlapping metamodel definitions and will be used to ensure conformance for the underlying models. A mapping family σ is built from the empty mapping family $\emptyset$.

Default objects have to be created of a property's type if the lower bound of its multiplicity is unequal to zero. This is required to preserve conformance. It is irrelevant if the property has a class type or primitive type. A property implicitly has a multiplicity of `0..1` if no multiplicity is stated. δ contains tuples of the form $\langle Nc, Np, Nat \rangle$ which maintain information of what objects that have to be created in the existing models as a consequence of new properties. Nc is the name of the class whose objects will contain or refer objects of Np's type[1]. The number of objects that need to be created is described by the natural number Nat, e.g. a property with multiplicity `3..4` requires the creation of three default objects to preserve conformance.

The adaptation analysis of a syntactic construct D is formalised by a deductive system for judgements $\langle \mathcal{E}, \sigma, \delta \rangle \vdash D \ \langle \mathcal{E}', \sigma', \delta' \rangle$, where $\langle \mathcal{E}, \sigma, \delta \rangle$ is the analysis environment *before* and $\langle \mathcal{E}', \sigma', \delta' \rangle$ is the environment *after* the analysis of D, where $\mathcal{E}'$ represents a package containing the derived metamodel. For updating the analysis environment, we use the associative operator + on mappings with the identity element $\emptyset$. Let $\mathcal{E} + \mathcal{E}'$ denote $\mathcal{E}$ *overriden* by $\mathcal{E}'$. Mappings are now formally defined.

Definition 3. *Let* n *be a name,* d *a declaration,* $i \in I$ *a mapping index, and* $[n \overset{c}{\mapsto}_i d]$ *the binding of* n *to* d *indexed by* i*. A mapping family* σ *is built from the empty mapping family* $\emptyset$ *and indexed bindings by the constructor* $+$*. The extraction of an indexed mapping* σ_i *from* σ *and application for the mapping* $\mathcal{E}$*, are defined as follows*

$$\begin{array}{lcl} \emptyset_i & = & \varepsilon \\ (\sigma + [n \overset{c}{\mapsto}_{i'} d])_i & = & \textbf{if } i = i' \textbf{ then } \sigma_i + [n \overset{c}{\mapsto}_i d] \\ & & \textbf{else } \sigma_i \\ \varepsilon(n) & = & \bot \\ (\mathcal{E} + [n \mapsto d])(n') & = & \textbf{if } n = n' \textbf{ then } d \\ & & \textbf{else } \mathcal{E}(n') \end{array}$$

Assume given two well-typed metamodels, $\mathcal{MM}_1$ and $\mathcal{MM}_2$. Since the metamodels are from two different domains, we assume unique class names (otherwise, this can easily be resolved by package annotations on class names). The environment $\mathcal{E}$ will initially contain well-typed class definitions from $\mathcal{MM}_1$ and $\mathcal{MM}_2$ and through a series of *adaptation operations*, we adapt or compose $\mathcal{MM}_1$ and $\mathcal{MM}_2$, causing modifications to $\mathcal{E}$ and σ, and additions to δ. The final environment that is returned after the analysis is the adapted metamodel(s) and the mappings constructed during the analysis for resolving name conflicts (i.e. substitutions generated by the framework) in addition to a specification of objects that need to be created due to property multiplicities with a non-zero lower bound.

[1]The exact realisation of object containment is dependent on the underlying implementation.

(ADD CLASS)

$$\frac{\mathcal{E}(C) = \bot \qquad \mathcal{E}' = \mathcal{E} + [C \mapsto \langle \varepsilon, \emptyset, \emptyset \rangle] \qquad \mathcal{E}', \sigma, \delta \vdash addProp(C, \overline{P}) \cdot addOp(C, \overline{O}) \langle \mathcal{E}', \sigma, \delta' \rangle}{\mathcal{E}, \sigma, \delta \vdash addClass(\langle C, \varepsilon, \overline{P}, \overline{O} \rangle) \langle \mathcal{E}', \sigma, \delta \rangle}$$

(CLASS MERGE)

$$\frac{\begin{array}{c}\mathcal{E}(C) = \langle C, \overline{B}, \overline{P}, \overline{O} \rangle \qquad \mathcal{E}(C') = \langle C', \overline{B'}, \overline{P'}, \overline{O'} \rangle \qquad C \neq C' \\ fresh(D) \qquad \sigma' = \sigma + [C \mapsto_c D] + [C' \mapsto_c D] \\ \sigma_p = anProp(\mathcal{E}, C, C') \qquad \sigma_o = anOp(\mathcal{E}, \sigma', C, C') \qquad \sigma'' = \sigma' + \sigma_p + \sigma_o \\ \mathcal{E}' = [(\mathcal{E} \setminus \{C, C'\})]_{\sigma''} + [D \mapsto \langle D, \overline{B}; \overline{B'}, [\overline{P}; \overline{P'}]_{\sigma''}, [\overline{O}; \overline{O'}]_{\sigma''} \rangle] \\ \delta' = createInst(C, [\overline{P'}]_{\sigma''}; flatten(\mathcal{E}', \overline{B'})) \qquad \delta'' = createInst(C', [\overline{P}]_{\sigma''}; flatten(\mathcal{E}', \overline{B}))\end{array}}{\mathcal{E}, \sigma, \delta \vdash merge(C, C') \langle \mathcal{E}', \sigma'', \delta + \delta' + \delta'' \rangle}$$

(ADD SUPERCLASS)

$$\frac{\begin{array}{c}\mathcal{E}(C') \neq \bot \qquad C \neq C' \qquad \mathcal{E}(C) = \langle C, \overline{B}, \overline{P}, \overline{O} \rangle \qquad \mathcal{E}(C') = \langle C', \overline{B'}, \overline{P'}, \overline{O'} \rangle \\ \sigma_p = anProp(\mathcal{E}, C, C') \qquad \sigma_o = anOpInh(\mathcal{E}, \varepsilon, C, C') \\ \sigma' = \sigma_p + \sigma_o \qquad \delta' = createInst(C, \overline{P'}; flatten(\mathcal{E}_{\sigma'}, \overline{B'}))\end{array}}{\mathcal{E}, \sigma, \delta \vdash addSuperClass(C, C') \langle [\mathcal{E}]_{\sigma'} + C \mapsto \langle C, C'; \overline{B}, [\overline{P}]_{\sigma_p}, [\overline{O}]_{\sigma_o} \rangle, \sigma + \sigma', \delta + \delta' \rangle}$$

(ADD INTERFACE CLASS)

$$\frac{\begin{array}{c}isPrimitive(\overline{P}) \qquad \mathcal{E}, \sigma, \delta \vdash addClass(\langle C, \varepsilon, \overline{P}, \overline{O} \rangle) \langle \mathcal{E}', \sigma, \delta \rangle \qquad opposite(\mathcal{R}) \\ \forall (C', \overline{P''}) \in \mathcal{R} \cdot \ \mathcal{E}'(C') = \langle C', \overline{B}, \overline{P'}, \overline{O'} \rangle \wedge \overline{P''}.Np \cap \overline{P'}.Np = \emptyset \wedge createInst(C', \overline{P''}) = \delta'' \\ uniqueProp(\mathcal{E}', \overline{P''}, \overline{B}) \qquad \mathcal{E}'' = \bigcup [C' \mapsto \langle C', \overline{B}, \overline{P'}; \overline{P''}, \overline{O'} \rangle] \qquad \delta''' = \bigcup \delta''\end{array}}{\mathcal{E}, \sigma, \delta \vdash addInterfaceClass(\langle C, \varepsilon, \overline{P}, \overline{O} \rangle, \mathcal{R}) \langle \mathcal{E}' + \mathcal{E}'', \sigma, \delta + \delta''' \rangle}$$

(CLASS OVERRIDE)

$$\frac{\begin{array}{c}\mathcal{E}(C) = \langle C, \overline{B}, \overline{P}, \overline{O} \rangle \qquad \mathcal{E}(C') = \langle C', \overline{B'}, \overline{P'}, \overline{O'} \rangle \qquad \forall P \in \overline{P} \cdot P \in \overline{P'} \qquad \forall O \in \overline{O} \cdot O \in \overline{O'} \\ \sigma' = [C \mapsto_c C'] \qquad \sigma_p = anProp(\mathcal{E}, C', \overline{B}) \qquad \sigma_o = anOpInh(\mathcal{E}, \sigma', C', \overline{B}) \\ \sigma'' = \sigma' + \sigma_p + \sigma_o \qquad \mathcal{E}' = [(\mathcal{E} \setminus C)]_{\sigma''} + [C' \mapsto \langle C', \overline{B}; \overline{B'}, [\overline{P'}]_{\sigma''}, [\overline{O'}]_{\sigma''} \rangle] \\ createInst(C, [\overline{P'} \setminus \overline{P}]_{\sigma''}; flatten(\mathcal{E}', \overline{B'})) = \delta' \qquad createInst(C', flatten(\mathcal{E}', \overline{B})) = \delta''\end{array}}{\mathcal{E}, \sigma, \delta \vdash override(C, C') \langle \mathcal{E}', \sigma + \sigma'', \delta + \delta' + \delta'' \rangle}$$

Figure 5: Definitions of the adaptation operations (rules) for class merging and additions. C and C' are class names.

(ADD PROPERTY)

$$\frac{\begin{array}{c}isPrimitive(t) \vee \mathcal{E}(t) \neq \bot \qquad \mathcal{E}(C) = \langle C, \overline{B}, \overline{P}, \overline{O} \rangle \qquad uniqueProp(\mathcal{E}, P, C) \\ P.Opp = \varepsilon \qquad \mathcal{E}' = \mathcal{E} + [C \mapsto \langle C, \overline{B}, P; \overline{P}, \overline{O} \rangle] \qquad createInst(C, P) = \delta'\end{array}}{\mathcal{E}, \sigma, \delta \vdash addProp(C, P) \langle \mathcal{E}', \sigma, \delta + \delta' \rangle}$$

(ADD BI-DIRECTIONAL PROPERTY)

$$\frac{\begin{array}{c}P.Type = C' \qquad \mathcal{E}(C) = \langle C, \overline{B}, \overline{P}, \overline{O} \rangle \qquad uniqueProp(\mathcal{E}, P, C) \\ P'.Type = C \qquad \mathcal{E}(C') = \langle C', \overline{B'}, \overline{P'}, \overline{O'} \rangle \qquad uniqueProp(\mathcal{E}, P', C') \\ P.Opp = P'.Np \qquad P'.Opp = P.Np \qquad \mathcal{E}' = \mathcal{E} + [C \mapsto \langle C, \overline{B}, P; \overline{P}, \overline{O} \rangle] + [C' \mapsto \langle C', \overline{B'}, P'; \overline{P'}, \overline{O'} \rangle] \\ createInst(C, P) = \delta' \qquad createInst(C', P') = \delta''\end{array}}{\mathcal{E}, \sigma, \delta \vdash addBiProp(C, P, C', P') \rangle \langle \mathcal{E}', \sigma, \delta + \delta' + \delta'' \rangle}$$

(ADD OPERATION)

$$\frac{\begin{array}{c}\forall t' \in \{t; O.Param\} \cdot \ isPrimitive(t') \vee \mathcal{E}(t') \neq \bot \qquad \mathcal{E}(C) = \langle C, \overline{B}, \overline{P}, \overline{O} \rangle \\ uniqueOp(\mathcal{E}, O, C) \qquad \mathcal{E}' = \mathcal{E} + [C \mapsto \langle C, \overline{B}, \overline{P}, O; \overline{O} \rangle]\end{array}}{\mathcal{E}, \sigma, \delta \vdash addOp(C, O) \langle \mathcal{E}', \sigma, \delta \rangle}$$

Figure 6: Adaptation operations for adding new properties and operations. P and O are property and operation definitions and C is used for class names.

3.2 The Adaptation Rules

Metamodels are adapted given an adaptation strategy Φ, $\mathcal{E}$ of intitial metamodel definitions, an empty mapping σ, and an empty set of object creation specifications δ. Thus if $\mathcal{E}, \emptyset, \emptyset \vdash \Phi \ \langle \mathcal{E}', \sigma', \delta' \rangle$, then we derive the resulting environment $\langle \mathcal{E}', \sigma', \delta' \rangle$. The adaptation rules are given in Figures 5 and 6. The involved operations are applied in a sequential manner as illustrated in Rule (SEQ):

$$
\begin{array}{ll}
anProp(\mathcal{E},C;\overline{C},\overline{B}) = & an(\mathcal{E},C,\mathcal{E}(C).Prop,\overline{B}) + anProp(\mathcal{E},\overline{C};\mathcal{E}(C).Inh,\overline{B}) \\
an(\mathcal{E},C,P;\overline{P},\overline{B}) = & an(\mathcal{E},C,P,\overline{B}) + an(\mathcal{E},C,\overline{P},\overline{B}) \\
an(\mathcal{E},C,P,\varepsilon) = & \varepsilon \\
an(\mathcal{E},C,(t,P,m,c,P_i),B;\overline{B}) = & \text{if } \mathcal{E}(B).Prop(P) \neq \emptyset \text{ then } [P \overset{C}{\mapsto}_{\rho} P'] \wedge fresh(P') \\
 & \text{else } an(\mathcal{E},C,(t,P,m,c,P_i),\overline{B};\mathcal{E}(B).Inh) \\
 & \\
anOp(\mathcal{E},\sigma,C,C') = & anMerge(\mathcal{E},\sigma,C',C,\mathcal{E}(C).Op,C') + anOpInh(\mathcal{E},\sigma,\mathcal{E}(C).Inh,C') \\
anOpInh(\mathcal{E},\sigma,C;\overline{C},\overline{B}) = & anCl(\mathcal{E},\sigma,C,\mathcal{E}(C).Op,\overline{B}) + anOpInh(\mathcal{E},\sigma,\overline{C};\mathcal{E}(C).Inh,\overline{B}) \\
anCl(\mathcal{E},\sigma,C,O;\overline{O},\overline{B}) = & anCl(\mathcal{E},\sigma,C,O,\overline{B}) + anCl(\mathcal{E},\sigma,C,\overline{O},\overline{B}) \\
anCl(\mathcal{E},\sigma,C,O,\varepsilon) = & \varepsilon \\
anCl(\mathcal{E},\sigma,C,(t,O,(\overline{t_iP_i})),B;\overline{B}) = & \text{if } [type(\mathcal{E}(B).Op(O).Param)]_\sigma = [\overline{t_i}]_\sigma \text{ then} \\
 & \quad \text{if } [\mathcal{E}(B).Op(O).Type]_\sigma = [t]_\sigma \text{ then } \varepsilon \\
 & \quad \text{else } [O \overset{C}{\mapsto}_{o} O'] \wedge fresh(O') \\
 & \text{else } anCl(\mathcal{E},\sigma,C,(t,O,(\overline{t_iP_i})),\overline{B};\mathcal{E}(B).Inh) \\
anMerge(\mathcal{E},\sigma,C',C,O;\overline{O},\overline{B}) = & anMerge(\mathcal{E},\sigma,C',C,O,\overline{B}) + anMerge(\mathcal{E},\sigma,C',C,\overline{O},\overline{B}) \\
anMerge(\mathcal{E},\sigma,C',C,O,\varepsilon) = & \varepsilon \\
anMerge(\mathcal{E},\sigma,C',C,(t,O,(\overline{t_iP_i})),B;\overline{B}) = & \text{if } [type(\mathcal{E}(B).Op(O).Param)]_\sigma = [\overline{t_i}]_\sigma \text{ then} \\
 & \quad \text{if } B = C' \text{then } [O \overset{S}{\mapsto}_{o} O'] \wedge fresh(O') \\
 & \quad \text{else if } [\mathcal{E}(B).Op(O).Type]_\sigma = [t]_\sigma \text{ then } \varepsilon \\
 & \quad \text{else } [O \overset{S}{\mapsto}_{o} O'] \wedge fresh(O') \\
 & \text{else } anMerge(\mathcal{E},\sigma,C',C,(t,O,(\overline{t_iP_i})),\overline{B};\mathcal{E}(B).Inh)
\end{array}
$$

Figure 7: Definition of the auxiliary functions *anOp* and *anProp*.

$$
\frac{\mathcal{E},\sigma,\delta \vdash [\varphi_1]_\sigma \langle \mathcal{E}_1,\sigma_1,\delta_1\rangle \quad \mathcal{E}_1,\sigma_1,\delta_1 \vdash [\overline{\varphi_2}]_{\sigma_1} \langle \mathcal{E}_2,\sigma_2,\delta_2\rangle}{\mathcal{E},\sigma,\delta \vdash \varphi_1 \cdot \overline{\varphi_2} \langle \mathcal{E}_2,\sigma_2,\delta_2\rangle} \text{(SEQ)}
$$

The analysis environment propagates throughout the analysis, i.e. $\overline{\varphi_2}$ is analysed in the context resulting from the analysis of φ_1. Note that the substitution, denoted $[\varphi]_\sigma$, rewrites φ into normal form in Rule (SEQ). σ is accumulated through sequential analysis of the adaptation operations, thus the substitution is deterministic; the rule substitutes input class, property and operation names by mappings in σ before retrieving or updating class information and thus ensures that class definitions from the latest environment are used in each basic rule application. By transitivity a sequence of class mergings is possible, e.g. $[X]_{[X\mapsto_c Y]+[Y\mapsto_c Z]+[Z\mapsto_c V]}$ maps the class named X to V. Rules for metamodel composition and class addition are given in Figure 5. In Rule (ADD CLASS) the analysis environment is extended with a new class definition that is previously not defined. Rule (CLASS MERGE) merges two classes C and C' into a class given a fresh name D, represented by *fresh(D)*. The new class will replace C and C' in the analysis environment, hence the rule redirects references by creating mappings from the old classes to D. We assume that metamodels are well-typed prior to adaptation. However, it is likely that overlapping definitions occur when classes are merged, which introduce type errors. Consequently, the rule will consider and *resolve* overlapping definitions that exist between the classes, i.e. C and C', and their superclasses. The auxiliary functions *anProp* and *anOp* (given in Figure 7) traverse the inheritance tree and resolve conflicts so that all property names are unique within the resulting class hierarchy, and that equally named operations with the same input types have the same return type. Mappings containing information of resolved conflicts are returned from the application of the rule (σ''). Finally, the class definition for D is constructed by specifying the superclasses of both C and C', and adding sets of conflict-free properties and operations, i.e. $[\overline{P};\overline{P'}]_{\sigma''}$ and $[\overline{O};\overline{O'}]_{\sigma''}$, respectively. The class D is added to $\mathcal{E}'$ whereas C and C' are removed. As other classes in $\mathcal{E}$ may still have old references to C and C', these classes must also be updated. Consequently, we apply the substitution mapping σ'' on $\mathcal{E}$ when creating the new environment (e.g. $\langle N,\bar{I},(C',P,m,c,P_i);\overline{P},\overline{O}\rangle_{[C'\mapsto_c D]} \rightarrow \langle N,\bar{I},(D,P,m,c,P_i);\overline{P},\overline{O}\rangle$). Moreover, σ'' also contains information of resolved conflicts in superclasses, thus these superclasses must also be updated to ensure well-typedness of the modified class hierarchy. *createInst* is a function that builds specifications of default objects that need to be created to preserve conformance. We have to ensure that existing models with objects of C or C' will have required objects as imposed by properties with a lower bound unequal to zero as reachable through D. That is, properties of D, $\overline{B}$ and $\overline{B'}$. *createInst* does only specify the smallest number of required default objects needed to ensure conformance. *flatten* returns a set of all properties found in the set of classes given as argument and their respective superclasses. The environment after the application of (CLASS MERGE) is $\langle \mathcal{E}',\sigma'',\delta+\delta'+\delta''\rangle$. Note that we can not merge attributes with the same names because these attributes may not represent the same concept from an ontological perspective, i.e. the attributes may describe different domain entities.

Rule (ADD SUPERCLASS) extends class C with a new superclass C' and overrides the old class definition. Similar to Rule (CLASS MERGE), the class hierarchy is traversed and conflicting definitions are resolved. The

resulting environment contains an updated version of C that replaces the previous version of the class. In particular, C additionally inherits from C'.

Rule (ADD INTERFACE CLASS) creates a new class (by Rule (CLASS UNION)) and constructs references between this and existing classes in the environment. *opposite* verifies that bi-directional relationships are correctly specified in $\mathcal{R}$. As there are no name conflicts, the mapping σ remains unchanged after the analysis. Rule (CLASS OVERRIDE) allows a class C' to override a class C if C' subsumes C. Finally, rules for extending classes with new operations and properties are given in Figure 6.

As mentioned, since operations *addClass*, *addInterfaceClass*, *addProp*, *addOp* and *addBiProp* add new metaelements, there are no existing models that refer to these elements. Consequently σ, which is later used to update existing models, remains unchanged for these operations. Rule (ADD PROPERTY) adds a new primitive- or class-typed property to an existing class. In contrast to Rule (ADD BI-DIRECTIONAL PROPERTY), this rule addresses properties without an opposite relation. Rule (ADD BI-DIRECTIONAL PROPERTY) adds a bi-directional relationship between two classes and verifies that its constituent uni-directional relationships are correct. Rule (ADD OPERATION) adds a new operation to a class. For the rules in Figure 6, *uniqueProp* ensures that the new property is not previously defined within super- or subclasses in the class hierarchy and *uniqueOp* verifies correct overloading of the new operation. See the requirements for well-typedness stated in Section 3. Notice that we do not resolve conflicts when new structure is added to the environment. As before, *createInst* creates a set of specifications for objects that need to be created to ensure conformance.

3.3 Example Revisited

We revisit the example introduced in Sec. 2 and show how the analysis framework addresses composition of the metamodels, resulting in the metamodel in Figure 3. We show the intermediate analysis steps and use the following abbreviations for class names: M for Merchandise, ROS for RightsOwnership, BC for BusinessContext and PS for ProfitShare. In the example we will consider the adaptation strategy given in Figure 8. The adaptation strategy comprises two operations: a *merge* operation and an *addInterfaceClass* operation.

$$\Phi = merge(M, BC) \cdot addInterfaceClass(\langle PS, \varepsilon, \langle (Real, percent, (1,1)) \rangle, \varepsilon\rangle, (M, \langle (PS, profitShares, (1,*)) \rangle), (PS, \langle (ROS, stakeholders, (1,*)) \rangle))$$

Figure 8: The adaptation strategy used in the example.

$$\mathcal{E} = \langle \langle M, \varepsilon, \langle (String, id) \rangle, \langle (String, format) \rangle \rangle, \langle BC, \varepsilon, \langle (String, id), (String, region), (String, country, (1,1)) \rangle, \langle (String, format) \rangle \rangle, \langle ROS, \varepsilon, \langle (Real, value), (String, firm), (String, country), \varepsilon \rangle, ... \rangle$$
$$\sigma = \delta = \emptyset$$

Figure 9: The initial environment.

Figure 9 gives the initial environment which consists of all classes from the two metamodels, e.g. the M, BC and ROS classes. As can be read from the figure, M has no superclass, an attribute id of type String and an operation format with type String as well. The BC class has no superclass, three attributes and one operation. Notice the lower bound in the multiplicity for the country attribute, which is 1. This means that an object of the BC class needs to have a value for this attribute. The other two attributes in the class have a 0..1 multiplicity by default. Finally, the ROS class has no superclass and three attributes.

The first operation in the adaptation strategy specifies that the M and BC classes should be merged. Merging the two classes yields a new class with the name MBC[2]. Merging of classes requires collecting information about the constituent classes of the merging operation. In this case, two mappings are created and stored, see Figure 10.

$$\mathcal{E}, \sigma, \delta \vdash merge(M, BC) \langle \mathcal{E}', \sigma', \delta' \rangle$$
$$\mathcal{E}' = \langle \langle MBC, \varepsilon, \langle (String, \#id), (String, id), (String, region), (String, country, (1,1)) \rangle, \langle (String, \#format), (String, format) \rangle \rangle, \langle ROS, \varepsilon, \langle (Real, value), (String, firm), (String, country) \rangle, \varepsilon \rangle, ... \rangle$$
$$\sigma' = [M \mapsto_c MBC] + [BC \mapsto_c MBC] + [id \overset{M}{\mapsto}_p \#id] + [format \overset{M}{\mapsto}_o \#format]$$
$$\delta' = \langle \langle M, country, 1 \rangle \rangle$$

Figure 10: The result of analysing the merge operation.

The first mapping binds the name M with MBC and the other mapping binds the name BC with MBC. Furthermore, equally named properties in the two classes need to be addressed, otherwise, this will result in a name conflict. The classes M and BC both have a property named id. Hence, the id property in the M class is given the new name #id by the *anProp* function (#id represents the new name for the id property). This renaming is also reflected by creating a new name mapping. The two classes also have an equally named operation. This name conflict is handled in a similar manner as for the id property. We use the notation $id \overset{M}{\mapsto}_p \#id$ for specifying that the name of the property *id* is mapped to the name *#id* in the M class.

Notice how an object creation specification is constructed since BC has an attribute country whose lower bound is 1. Hence, existing product models need to

[2]The name chosen is decided by the implementation.

be updated with a value for this attribute in objects of MBC (previously M). Finally, a new environment is constructed and superfluous classes, that is M and BC, are removed. Note that the new environment still contains the remaining unaffected classes of the two metamodels.

The next step is analysis of the *addInterfaceClass* operation, refer Figure 8 for details on how the *addInterfaceClass* operation is specified. First, a new class named PS is added to the environment. The purpose of this class is to bridge the MBC and RightsOwnership classes, see Figure 3. Second, a selection of classes in the environment is revised. Specifically, the classes relating to the PS class, and potentially the PS class itself, are updated with class references. Notice how the accumulated mapping of $[M \mapsto_c MBC]$ has been applied (by SEQ) before the analysis by the (ADD INTERFACE CLASS) rule. Thus, even though the *addInterfaceClass* operation refers to M, the name mapping ensures that the MBC class is used in the analysis. Also, pay note of how the *createInst* function is applied to specify generation of default objects of the PS and ROS classes, which is required since these classes are related to by the references profitShares and stakeholders, respectively. Both of these references have a non-zero lower bound in their multiplicity.

$$\mathcal{E}', \sigma', \delta' \vdash addInterfaceClass(\langle PS,\ \varepsilon,\ \langle (Real,\ percent)\rangle, \varepsilon\rangle,$$
$$((MBC,\ \langle (PS,\ profitShares,\ (1,*))\rangle),$$
$$(PS, \langle (ROS,\ stakeholders,\ (1,*))\rangle)\rangle \langle \mathcal{E}'', \sigma'', \delta''\rangle$$

$$createInst(MBC, \langle (PS,\ profitShares,\ (1,*))\rangle) = \langle MBC, profitShares, 1\rangle$$
$$createInst(PS, \langle (ROS,\ stakeholders,\ (1,*))\rangle) = \langle PS, stakeholders, 1\rangle$$
$$\mathcal{E}'' = \langle \langle MBC, \varepsilon, \langle (String,\ id), (String,\ \#id), (String,\ region),$$
$$(String,\ country, (1,1)), (PS,\ profitShares,\ (1,*))\rangle,$$
$$\langle (String,\ \#format), (String,\ format)\rangle\rangle,$$
$$\langle ROS, \varepsilon, \langle (Real,\ value), (String,\ firm), (String,\ country),$$
$$(PS,\ stakeholders,\ (1,*))\rangle, \varepsilon\rangle, \langle PS,\ \varepsilon,\ \langle (Real,\ percent)\rangle, \varepsilon\rangle, ...\rangle$$

$$\sigma'' = [M \mapsto_c MBC] + [BC \mapsto_c MBC] + [id \overset{M}{\mapsto}_p \#id] + [format \overset{M}{\mapsto}_o \#format]$$
$$\delta'' = \langle \langle M, country, 1\rangle, \langle MBC, profitShares, 1\rangle, \langle PS, stakeholders, 1\rangle\rangle$$

Figure 11: The final environment, mappings and object creation specifications.

The final environment, $\langle \mathcal{E}'', \sigma'', \delta''\rangle$, can be seen in Figure 11. The enviroment $\mathcal{E}''$ corresponds to the metamodel in Figure 3 and contains updated versions of the classes MBC and ROS, the new PS class, and all remaining unaltered classes of the constituent metamodels. δ has been updated to indicate that a default object has to be created by both PS (profitShares is typed with PS) and ROS (stakeholders is typed with ROS) for all existing models that contain objects of MBC (M and BC). Notice that it does not exist any models with objects of PS (since this is a new class), however, since MBC has a property typed with PS whose lower bound is 1, this requires creation of an object of PS in all existing models previously conforming to either of the two metamodels. This in turn requires creation of default objects of ROS since PS has a property typed with ROS whose lower bound is 1 as well.

4 CONFORMANCE

We use the definition of conformance as given in (Henderson-Sellers, 2012). A model $\mathcal{M}$ conforms to a metamodel $\mathcal{MM}$ if all objects in the model are classified by one concept in the metamodel. A concept is described by a class. Γ denotes a classification abstraction. The extension of a concept, $\varepsilon(C)$, is defined by a set of predicates that characterise the same elements. The set of these predicates is known as the intension of the concept - $\iota(C)$.

Definition 4. *A model $\mathcal{M}$ conforms to a metamodel $\mathcal{MM}$, denoted conformsTo($\mathcal{M}, \mathcal{MM}$), iff. $\mathcal{MM}$ is well-typed and:*

$$\forall_i o_i \in \mathcal{M}, \exists_j C_j \in \mathcal{MM} \text{ such that } o_i\ \Gamma\ C_j, \text{ where}$$
$$o_i\ \Gamma\ C_j \Leftrightarrow o_i \in \varepsilon(C_j)\text{, and}$$
$$\varepsilon(C) = \{x | P(x)\} \text{ where } P = \iota(C)$$

Based on the definition of conformance we have the following property:

Lemma 1. *Let $\mathcal{MM}_1$ and $\mathcal{MM}_2$ be well-typed metamodels, $\mathcal{M}$ a model, and $\mathcal{E}$ contains all classes in $\mathcal{MM}_1$ and $\mathcal{MM}_2$. If conformsTo($\mathcal{M}, \mathcal{MM}_1$), then conformsTo($\mathcal{M}, \mathcal{E}$). Similarly, if conformsTo($\mathcal{M}, \mathcal{MM}_2$), then conformsTo($\mathcal{M}, \mathcal{E}$).*

As the considered adaptation operations do not remove classes, operations or properties, or change their type declarations, well-typedness is preserved for the resulting metamodel.

We show that with an adaptation strategy Φ for metamodel adaptation and the accumulated effects σ and δ, the associated models can be updated so that they conform to the altered metamodel. Figure 12 illustrates this for the *merge* operation. The figure contains a model of each of the metamodels given in Figure 4. The circles illustrate model objects. Notice how default objects are created to satisfy conformance according to non-zero lower bound multiplicities in the metamodels. Thus, two default objects are required to be added to the M_a model for this model to conform to the MM_{1-2} metamodel. Similarily, a default object has to be added to the M_b model for this to conform to MM_{1-2}.

The semantics of model evolution is described by the *transform* function, see Figure 13. Let

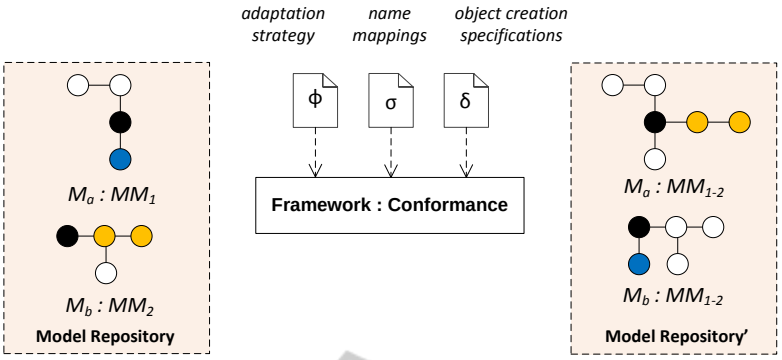

Figure 12: Illustration of how conformance is re-established.

$proj_c(\sigma_p)$ return property mappings for class C, i.e., $[Np \xrightarrow{c} Np]$ and $proj_c(\delta)$ object creation specifications, i.e. $\langle C, Np, Nat \rangle$. *createObj* produces objects as specified by δ. Object creation is only performed if the containing object does not already contain/refer these objects. A class may have a reference with a non-zero lower bound that is typed with a class that contains a reference with a non-zero lower bound to the first class. That is, there are two references between a pair of classes where both references have a non-zero lower bound. In this case, and in similar more comprehensive cases, the algorithm/function will not terminate. This may be addressed by reusing already created objects of the classes. *update* creates a new version of the given object. We show an excerpt of the function, the rest is defined in a similar manner.

Theorem 1. *Let $\mathcal{MM}_1$ and $\mathcal{MM}_2$ be well-typed metamodels, and let $\mathcal{E}$ contain all classes in $\mathcal{MM}_1$ and $\mathcal{MM}_2$, and $\mathcal{M}$ an arbitrary model such that conformsTo($\mathcal{M}, \mathcal{MM}_1$) or conformsTo($\mathcal{M}, \mathcal{MM}_2$). Let Φ be an adaptation strategy such that $\mathcal{E}, \emptyset, \emptyset \vdash \Phi \langle \mathcal{E}', \sigma, \delta \rangle$, then we have conformsTo(transform($\mathcal{M}, \Phi, \sigma, \delta$), $\mathcal{E}'$).*

Proof sketch. The proof is by induction over the length of the adaptation strategy Φ and then by cases. By Lemma 1 we have $conformsTo(\mathcal{M}, \mathcal{E})$ and show that if $\mathcal{E}, \emptyset, \emptyset \vdash \varphi_o \cdots \varphi_i \langle \mathcal{E}', \sigma, \delta \rangle$ and $conformsTo(transform(\mathcal{M}, \varphi_o \cdots \varphi_i, \sigma, \delta), \mathcal{E}')$ then if $\mathcal{E}', \sigma, \delta \vdash \varphi_{i+1} \langle \mathcal{E}'', \sigma', \delta' \rangle$, then $conformsTo(transform(\mathcal{M}, \varphi_o \cdots \varphi_{i+1}, \sigma', \delta'), \mathcal{E}'')$. Let o be an object in $\mathcal{M}$ and show that o remains a valid instance of its class after each adaptation step that modifies $\mathcal{E}$. We show that properties and operations are analysed which yields name mappings specified in σs. The mappings are used to update affected property name references (slots) in o. Hence, renamings, as a consequence of resolving name conflicts, are reflected upon the objects of these classes. Values for properties with a non-zero lower bound need to be created, and this is ensured by *createInst*, where the required type of objects and their respective counts are determined based on information in δs. □

```
transform(M, φ0 ··· φn, σ, δ ) : M
  let transform(φi ··· φn)
    for each object ⟨oid : N, (P,v)⟩ ∈ M do
      if φi == addClass(⟨C, ε, P, O⟩) // No action – no objects exist
      if φi == merge(C, C')
        σ' = projc(σc); projc'(σc)
        if N == C then
          update(M, ⟨oid : [C]projc(σc),
                createObj((P,v), projc(δ)); [(P,v)]projc(σp);σ'⟩)
        else if N == C' then
          update(M, ⟨oid : [C']projc'(σc),
                createObj((P,v), projc'(δ)); [(P,v)]projc'(σp);σ'⟩)
        else if (C subClassOf N) or (C' subClassOf N) then
          update(M, ⟨oid : N, [(P,v)]projN(σp);σ'⟩
        else update(M, ⟨oid : N, [(P,v)]σ'⟩)

        [φi+1 ··· φn]σ'

      if φi == addInterfaceClass(⟨C, ε, P, O⟩, (D, P') ∪ R)
        if N = D then
          update(M, ⟨oid : N, createObj(P', projD(δ)); (P,v)⟩)

      if φi == addProp(C, P)
        if N = C then
          update(M, ⟨oid : N, createObj(P, projN(δ)); (P,v)⟩)
    ...
    endfor
    tranform(φi+1 ··· φn)
  in
    transform(φ0 ··· φn)
  end
return M
```

Figure 13: The transform function.

4.1 Example Revisited

In Section 3.3 we showed how the product/merchandise metamodel and business context metamodel were combined by applying a simple adaptation strategy Φ involving merge and interfacing operations. Consequently, all existing product and business context models should now be considered as models of the composite metamodel. In order to re-establish conformance between existing models and the composite

metamodel, existing models must be updated as specified by the the mappings in σ'', and default object specifications in δ''.

Figure 14 shows how two existing models of the product and business context metamodels, respectively, are updated to ensure conformance with the composite metamodel. In particular, *transform*$(\mathcal{M}, \Phi, \sigma'', \delta'')$ updates objects of M and BC to be valid instances of MBC, values of the attribute id in M to be values of #id, and creates a default value for country in all objects that were previously instances of class M, as well as default objects for PS and ROS in all existing models. As the objects do not contain any references to operations, no changes are required for the renaming of the format operation in M. Clearly, we are aware that the default values and objects may have to be updated to reflect the domain, e.g. a correct value of country has to be specified. However, such considerations are outside the scope of our work.

5 RELATED WORK

One approach for automatic metamodel evolution is discussed in (Wachsmuth, 2007). The work is inspired by object-oriented refactoring, and is based on a stepwise adaptation of metamodels using transformations. Three different types of transformations are used to achieve this: refactoring (e.g. renaming of an element), construction (e.g. adding a new class or property) and destruction (e.g. removing a class or property). The paper also discusses how models may be co-adapted by defining transformations that work on models. The work of our paper resembles that of (Wachsmuth, 2007). We have defined a set of basic operations that can be combined arbitrarily to define adaptation strategies. We take things a step further by formally defining all operations and proving that conformance are preserved after adaptation. Also, our approach uses analysis as a prerequisite for supporting parallel evolution of an arbitrary number of metamodels.

(Ruscio et al., 2012) discusses how artefacts in the metamodelling ecosystem are impacted when metamodels evolve. In particular, the paper discusses how adaptation of artefacts should be facilitated inherently by the metamodel definition. This way, artefacts evolve directly as a consequence of metamodel evolution. It is argued that this perspective on co-evolution is important in order for metamodels to freely evolve without being constrained by required modifications to other artefacts, which may be both complex and expensive to perform. In our paper, we have seen how the accumulated effects can be used to update existing models. However, the effects are structured in a way that supports co-adaptation of other artefacts in the metamodelling ecosystem. Hence, our work addresses the concerns put forward in (Ruscio et al., 2012).

The proposed analysis of our paper can be supported by model migration processes such as the one described in (Gruschko et al., 2007). The adaptations to be analysed can be provided by a change recorder (Gruschko et al., 2007) and model migrations can be implemented using model transformation languages such as QVT(ObjectManagementGroup, 2007), ATL(Jouault, 2005) and ETL(Kolovos et al., 2006b)(Rose et al., 2010) where the analysis effects give mappings/rules for how model elements from a input model can be mapped to elements in a target model. When creating an element, a transformation may fail if a target metaclass is not defined (Gruschko et al., 2007). Our static analysis prevents this and ensures that execution of generated model transformations will succeed. Metamodel evolution can also be described using visual languages such as the one proposed in (Narayanan et al., 2009) where migration rules are defined with a graphical syntax.

(Cicchetti et al., 2008b)(Herrmannsdoerfer et al., 2009) discuss mechanisms for addressing co-evolution using difference models and transactions, respectively. The work of (Cicchetti et al., 2008a) discusses how a difference model can be used to automatically create transformations that support co-evolution of models. According to this work, a difference model conforms to an extended KM3 meta-metamodel. Specifically, the KM3 meta-metamodel has been revised with classes for expressing atomic modifications, e.g. *reference added*, *attribute changed* or *class added*. By substracting one metamodel from another it is possible to derive a difference model which gives all the changes required to derive the one metamodel from the other. The discussion differentiates between *parallel independent* and *parallel dependent* modifications. Parallel dependent modifications can, as the name suggests, not be performed in parallel since the modifications depend on each other. A difference model is decomposed into breaking resolvable and unresolvable changes. This in turn yields two types of transformations: transformations that can be automatically generated and transformations that must be completed with input from the user. An implementation is available for difference models containing parallel independent changes, whereas a solution for parallel dependent changes is sketched in the paper. The latter is addressed by an iterative process that splits the difference models into submodels that only contain par-

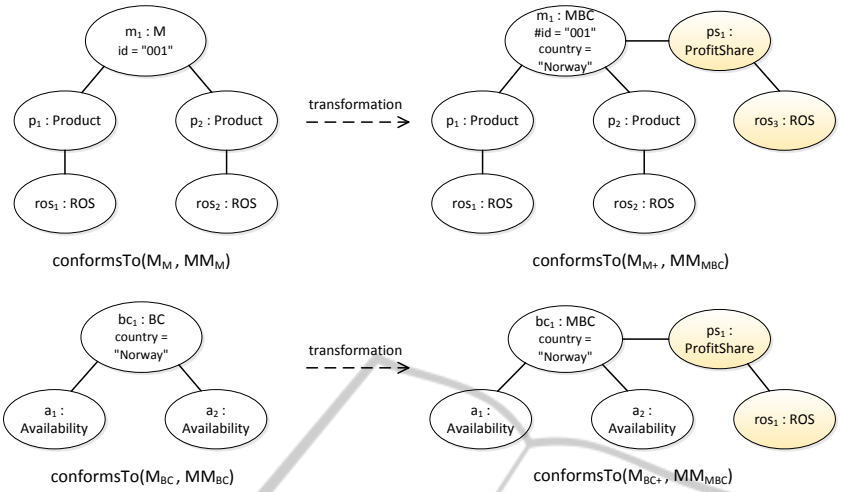

Figure 14: Evolution of models to ensure conformance.

allel independent changes. Our approach resembles that of (Cicchetti et al., 2008a). However, there are some significant differences. First, our approach derives effects directly from the analysis. The effects are used as input for the transform function or for generating model-to-model transformations; instead of a difference model. Second, we support adaptation operations that create dependencies between an arbitrary number of metamodels, yet ensure that conformance is preserved for all existing models of all metamodels. We have focused on breaking resolvable adaptations. We support both parallel independent and parallel dependent adaptation operations. The analysis verifies that dependent operations are applied in an appropriate sequence, as given by the user, which can be supported by an interactive editor.

The work of (Garcs et al., 2009) resembles that of (Cicchetti et al., 2008a). The main difference is how metamodels are matched using heuristics in order to derive the difference model.

(Taentzer et al., 2012) and (Mantz et al., 2010) formalise metamodel evolution and model co-evolution using category theory and graph transformations. In contrast, we formalise the low-level operations that typically emerge in metamodel/model co-evolution senarios based on MOF-metamodels which are the the most common metamodel formalisation used in industry. Hence, our approach is closer to existing frameworks and tools. We have not found any work that formalises metamodel/model co-evolution and metamodel composition in a similar manner as our approach.

6 CONCLUSION AND FUTURE WORK

In this paper we have discussed a mechanism for metamodel adaptation and showed how model conformance can be preserved when metamodels are adapted and modified, e.g. composed. We formalised metamodel adaptation operations and addressed co-evolution of models. We also proved that conformance can be re-established between models and their altered metamodels.

In the proposed framework, conflicts during merging of classes are remedied by renaming and not resolved from an ontological perspective. We foresee that considering ontological knowledge as part of the analysis would refine the adaptation mechanism, where adaptation of metamodels can be tuned by utilising domain knowledge, e.g. classes representing the same type of domain concepts may be merged, etc. This way we would also address matching of classes prior to merging.

We addressed how the accummulated effects can be used to update models. These effects can also be used to update existing tools and model-to-model transformations that are defined according to the constituent metamodels. This can be achieved by defining a similar algorithm as described for model co-evolution by the *transform* function. We have focused on model structure, and not considered adaptation of language semantics, which we consider as future work.

REFERENCES

Cicchetti, A., Di Ruscio, D., Eramo, R., and Pierantonio, A. (2008a). Automating co-evolution in model-driven

engineering. In *Proc. of the Enterprise Distrubuted Object Computing Conference (EDOC'08)*.

Cicchetti, A., Di Ruscio, D., Eramo, R., and Pierantonio, A. (2008b). Meta-model differences for supporting model co-evolution. In *Proc. of the 2nd International Workshop on Model-Driven Software Evolution.*

Del Fabro, M. D. and Valduriez, P. (2007). Semi-automatic model integration using matching transformations and weaving models. In *Proceedings of the 2007 ACM Symposium on Applied Computing, pp.963-970. ACM*.

Didonet Del Fabro, M., Bzivin, J., and Valduriez, P. (2006). Weaving models with the eclipse amw plugin.

Favre, J.-M. (2004). Towards a basic theory to model model driven engineering. In *3rd international workshop on Software Model Engineering (WISME '04)*.

Fleurey, F., Baudry, B., France, R., and Ghosh, S. (2008). A generic approach for automatic model composition. In *Models in Software Engineering, LNCS vol. 5002, pp.7-15. Springer*.

Garcs, K., Jouault, F., Cointe, P., and Bzivin, J. (2009). Managing model adaptatio by precise detection of metamodel changes. In *Proc. of the 5th European Conference on Model Driven Architecture*.

Gruschko, B., Kolovos, D., and Paige, R. (2007). Towards synchronizing models with evolving metamodels. In *Workshop on Model-Driven Software Evolution*.

Henderson-Sellers, B. (2012). *On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages*. Springer.

Herrmannsdoerfer, M., Benz, S., and Juergens, E. (2009). Cope: Coupled evolution of metamodels and models for the eclipse modeling framework. In *ECOOP 2009 - Object-Oriented Programming. LNCS vol. 5653, pp.52-76. Springer*.

Herrmannsdoerfer, M., D. Vermolen, S., and Wachsmuth, G. (2011). An extensive catalog of operators for the coupled evolution of metamodels and models. In *Software Language Engineering, LNCS vol. 6563, pp.163-182. Springer*.

Jouault, F.and Kurtev, I. (2005). Transforming models with atl. In *Proc. of the Model Transformations in Practice Workshop at MoDELS 2005. Springer-Verlag*.

Kent, S. (2002). Model driven engineering. In *Integrated Formal Methods, LNCS vol. 2335, pp.286-298. Springer*.

Kolovos, D., Paige, R., and Polack, F. (2006a). Merging models with the epsilon merging language (eml). In *Model Driven Engineering Languages and Systems, LNCS vol. 4199, pp.215-229. Springer*.

Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2006b). Eclipse development tools for epsilon. In *In Eclipse Summit Europe, Eclipse Modeling Symposium*.

Mantz, F., Rutle, A., Lamo, Y., Rossini, A., and Wolter, U. (2010). Towards a formal approach to metamodel evolution. In *Nordic Workshop on Programming Theory.*

Morin, B., Klein, J., and Barais, O. (2008). A generic weaver for supporting product lines. In *13th international workshop on Early Aspects (EA '08), pp.11-18. ACM Press*.

Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., and Jzquel, J.-M. (2009). Weaving variability into domain metamodels. In *Model Driven Engineering Languages and Systems, LNCS vol. 5795, pp.690-705. Springer*.

Narayanan, A., Levendovszky, T., Balasubramanian, D., and Karsai, G. (2009). Automatic domain model migration to manage metamodel evolution. In Schrr, A. and Selic, B., editors, *MoDELS*, volume 5795 of *LNCS*. Springer.

ObjectManagementGroup (2007). *MOF QVT Final Adopted Specification*. OMG.

ObjectManagementGroup (2014). Meta object facility (mof) core specification.

Rose, L. M., Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2010). Model migration with epsilon flock. In *Proc. of the Third International Conference on Theory and Practice of Model Transformations*, ICMT'10, pages 184–198. Springer-Verlag.

Rose, L. M., Paige, R. F., Kolovos, D. S., and Polack, F. A. C. (2009). An analysis of approaches to model migration. In *Proceedings of the Models and Evolution Workshop. ACM*.

Ruscio, D. D., Iovino, L., and Pierantonio, A. (2012). Evolutionary togetherness: How to manage coupled evolution in metamodeling ecosystems. In *In: Graph Transformations. LNCS vol. 7562, pp.20-37. Springer*.

Seidewitz, E. (2003). What models mean. In *IEEE Software, vol. 20, no. 5, pp.26-32.*

Steel, J. and Jzquel, J.-M. (2007). On model typing. In *Software and Systems Modeling, vol. 6, no. 4, pp.401-413. Springer*.

Taentzer, G., Mantz, F., and Lamo, Y. (2012). Cotransformation of graphs and type graphs with application to model co-evolution. In *In: Graph Transformations, LNCS vol. 7562, pp.326-340. Springer*.

TheEclipseFoundation (2014). Eclipse modeling framework (emf).

Wachsmuth, G. (2007). Metamodel adaptation and model co-adaptation. In *ECOOP 2007 - Object-Oriented Programming. LNCS vol. 4609, pp.600-624. Springer*.