Towards the flexible reuse of model transformations: A formal approach based on Graph Transformation

Juan de Lara[a], Esther Guerra[a]

[a] *Computer Science Department, Universidad Autónoma de Madrid (Spain)*

Abstract

Model transformations are the *heart and soul* of Model Driven Engineering (MDE). However, in order to increase the adoption of MDE by industry, techniques for developing model transformations in the large and raising the quality and productivity in their construction, like reusability, are still needed.

In previous works, we developed a reutilization approach for graph transformations based on the definition of *concepts*, which gather the structural requirements needed by meta-models to qualify for the transformations. Reusable transformations are typed by concepts, becoming *transformation templates*. Transformation templates are instantiated by *binding* the concept to a concrete meta-model, inducing a retyping of the transformation for the given meta-model.

This paper extends the approach allowing heterogeneities between the concept and the meta-model, thus increasing the reuse opportunities of transformation templates. Heterogeneities are resolved by using algebraic adapters which induce *both* a retyping and an adaptation of the transformation. As an alternative, the adapters can also be employed to induce an adaptation of the meta-model, and in this work we show the conditions for equivalence of both approaches to transformation reuse.

Keywords: Model-Driven Engineering, Graph Transformation, Meta-modelling, Genericity, Reusability

1. Introduction

Model-Driven Engineering (MDE) [3, 38] promotes an active use of models in the different phases of the software development. This involves the transformation of models between different languages – ranging from general-purpose to domain-specific modelling languages (DSMLs) – until code for the final application is generated.

MDE can be seen as a *reutilization* approach, where modelling languages and their associated transformations and code generators are reused across projects to describe different applications within a domain, but with certain variability that is configured through a model. However, it is also true that MDE is *type-centric* [7], because the different supporting artefacts (transformations and code generators) are defined over the types of a specific meta-model and cannot be reused for other meta-models, even if they share *essential* structural features. This rigidity hampers the

Email addresses: `Juan.deLara@uam.es` (Juan de Lara), `Esther.Guerra@uam.es` (Esther Guerra)

adoption of MDE in industry because similar transformations have to be repeatedly developed, even for meta-models with only slight differences.

Taking ideas from generic programming, in previous works we proposed the definition of transformations over so-called *concepts* [7, 16, 17], instead of over concrete meta-models. In our context, a concept specifies the structural requirements that meta-models need to fulfil in order to be able to apply a certain model operation (e.g. a transformation) on their instances. Our concepts resemble meta-models, but their elements (classes, references, fields) are variables that need to be *bound* to concrete meta-model elements. In this way, similar to generic programming templates [16], a *transformation template* is defined over a concept and is instantiated for a specific meta-model via a *binding*.

In [6], we formalized these techniques using graph transformation (GT) [11] to express model transformations, and restricting the binding to simple injective mappings between the concept and the meta-model elements. In this paper, we expand the formalization by allowing a more flexible binding by means of algebraic *adapters*, which are able to resolve heterogeneities between the concept and the meta-model, in the line of [29]. This approach increases the reuse opportunities of transformation templates because their associated concepts can be bound to a wider set of meta-models. Interestingly, the formalization of our adapters involves building a "virtual view" that unifies the two main approaches to genericity in MDE (namely, adaptation of the transformation [29] and meta-model adaptation [18]) and enables the study of the conditions for their equivalence.

The rest of this paper is organized as follows. Section 2 presents an overview of our approach. Section 3 formalizes (meta-)models and concepts. Section 4 introduces binding adapters and Section 5 shows their use to instantiate a GT template and to build a derived model. Section 6 compares with related work and, finally, Section 7 concludes the paper and identifies lines of future work. An appendix shows the details of the different proofs.

2. Motivation, Overview and Challenges

Assume we want to define a catalogue of refactorings for object-oriented notations [14] using GT rules. The first step is to define a meta-model so that the rules can be typed. However, this means that the rules will only be applicable to instances of such meta-model. This prevents the refactorings from being reused, as they cannot be applied to other object-oriented notations sharing common features – like UML class diagrams [37], KM3 [22] or Ecore [34] – but we need to encode slight variations of the same refactorings for each notation.

To overcome this limitation, we propose defining the rules over a so-called *concept*, as illustrated in Figure 1. Label 1 depicts a concept for the refactoring of object-oriented notations. Label 2 shows one simple refactoring rule, which moves a field from a class to one of its parents. This rule is typed over the concept. A concept has the form of a standard meta-model, but it needs to be bound to some concrete meta-model, as shown in label 3. This binding induces an adaptation of the rule via a high-order transformation (HOT) to make it applicable to the meta-model instances, as shown in label 4. Hence, similar to generic programming [10], GT rules so defined become *templates* that need to be instantiated for particular meta-models. This approach promotes reusability because the same transformation can be applied to every meta-model to which we can bind the concept. The figure shows part of the binding of the concept to a simplification of the UML2 meta-model [37] and to the Java meta-model (a slight modification of the one proposed by the *JaMoPP* project [19]).

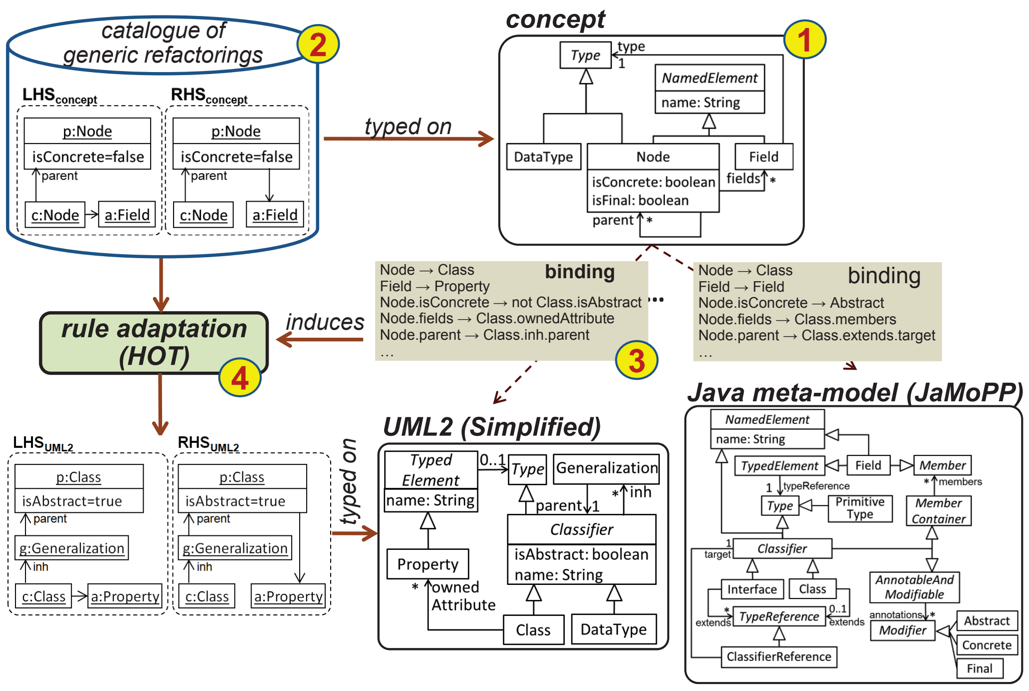


Figure 1: Rule template defined over a concept, and instantiation via a binding.

An additional advantage of this approach is that concepts are simpler, with less accidental details than the meta-models they get bound to, resulting in simpler transformations. This is so because concepts are specifically designed for a transformation while meta-models gather the abstractions of a domain, and hence may become large. In the running example, the concept contains the elements strictly needed to define object-oriented refactorings. The meta-models to which we bind the concept are much more complex, as they contain elements for other aspects which are not used by the transformation. For example, the UML2 superstructure meta-model contains over 330 classes, while *JaMoPP* has over 240.

In practice, a concept reflects some design decisions, so that the structural requirements gathered by it may have been implemented differently in concrete meta-models. For instance, the meaning of a field in the concept and the corresponding field in the meta-model may differ. This is the case of the *isConcrete* field that our example concept uses to represent whether a class is abstract or concrete: the UML2 meta-model uses the field *isAbstract* with opposite meaning for the same purpose, and *JaMoPP* uses the *marker* classes *Abstract* and *Concrete* instead. Some other times, the meta-models lack fields declared in the concept, like the field *isFinal* which is not defined by *Class* in the UML2 meta-model. Finally, there can be structural differences between the concept and the meta-models. For example, our concept represents inheritance with the association *parent*, but UML2 uses the intermediate class *Generalization*.

Figure 2 summarizes these heterogeneities. Hence, the challenge is to formally specify these heterogeneities, and being able to instantiate a GT template accordingly to obtain rules working on the bound meta-models. Figure 1 shows a rule template (label 2) and the desired rule instantiation for the binding to UML2 (label 4). The instantiated rule is typed over the UML2 meta-model and resolves the heterogeneities expressed in the binding (heterogeneities (a) and (c) in Figure 2).

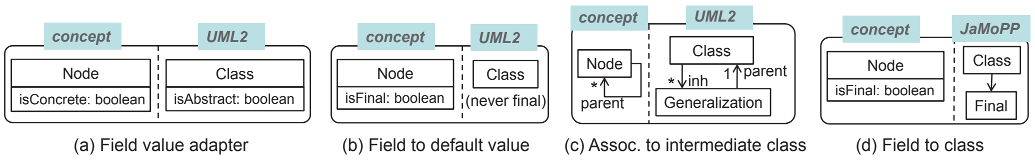


Figure 2: Some heterogeneities between the concept and concrete meta-models, to be resolved.

Instead of instantiating the template rules, a different way to achieve transformation reuse is to modify the concrete meta-model to make it "compatible" (a subtype) with the concept [18, 33]. In this second approach – meta-model adaptation – the elements that appear in the concept but not in the meta-model are added to the meta-model as derived elements that are calculated from queries on meta-model elements. Accordingly, the models to which we want to apply the transformation need to be modified as well. As an example, Figure 3 shows to the left a model M conformant to the UML2 meta-model, and to the right its extension M_s with the derived information present in the concept but not in the UML2 meta-model, namely, the derived attributes *isConcrete*, *isFinal* and *parent* in class *NodeClass*. If the modelling framework has automated mechanisms to synchronize the derived elements with the "regular" modelling elements through queries, then we can retype the original transformation and apply it to the instances of the adapted meta-model (like M_s in the figure), without adapting the transformation itself.

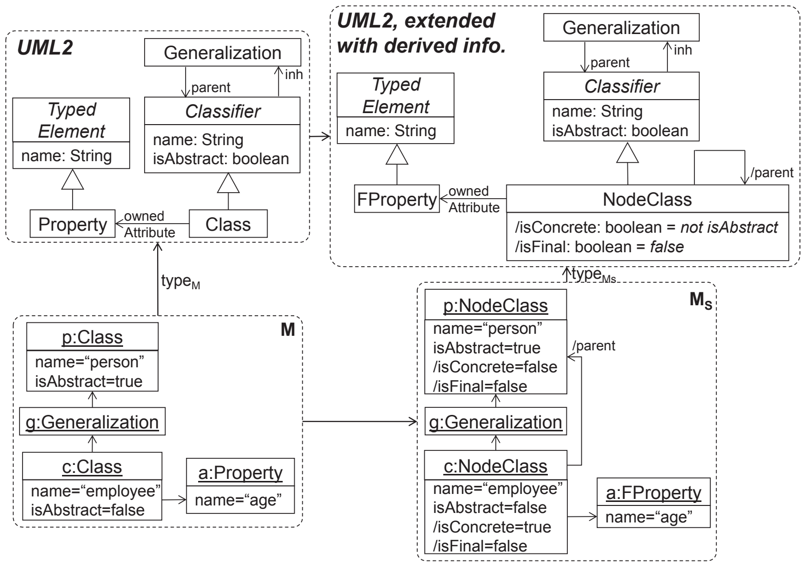


Figure 3: The UML2 meta-model and the model M are extended with derived information to enable rule application.

Hence, the following challenges remain in order to achieve the reutilization of transformations by means of concepts. First, in order to increase the reutilization oportunities, we need a flexible mechanism, which we call adapters, to resolve the heterogeneities between the concept and the concrete meta-model. Second, we need a mechanism to use adapters both for transformation adaptation, and for meta-model adaptation, ensuring that both kinds of adaptations are compatible.

In order to tackle these challenges, in sections 3–5 we present a unifying formalization where we define adapters to resolve heterogeneities between concepts and meta-models. Adapters can be used to both instantiate a GT template, and extend a meta-model and its instance models with the derived information needed to apply a GT template. In the latter case, the synchronization of derived and regular elements is emulated by means of rules acting on instances of the extended meta-model. Finally, in section 5.3 we show that, instantiating a template for a concrete meta-model preserves the behaviour of the original GT template, while being compatible with the meta-model adaptation approach.

3. Models, Meta-Models and Concepts, Algebraically

In this section we provide the formal basis to represent models, meta-models and concepts.

3.1. Models

The underlying graph model we use is that of node-attributed, typed graphs [11]. This kind of graphs represents data values as a special kind of nodes, and attributes as edges between graph nodes and data nodes.

Definition 1 (Graph)**.** *An attributed graph $G = \langle V; E; A; D; src_E, tar_E : E \to V; src_A : A \to V; tar_A : A \to D\rangle$, is made of a set V of vertices, a set E of edges, a set A of attributes, a set D of data nodes, and functions that return the source and target vertices of an edge (src_E, tar_E), and the owning vertex and data value of an attribute (src_A, tar_A).*

Example 1. Figure 4(a) shows a graph example, where the vertices (set V) are represented as rectangles, the data nodes (set D) are represented as rounded rectangles, the edges (set E) are represented as continuous arrows, and the attributes (set A) are represented as dotted arrows connecting the nodes as indicated by the src_A and tar_A functions. Figure 4(b) shows the same graph in a more compact notation, similar to the one for UML object diagrams.

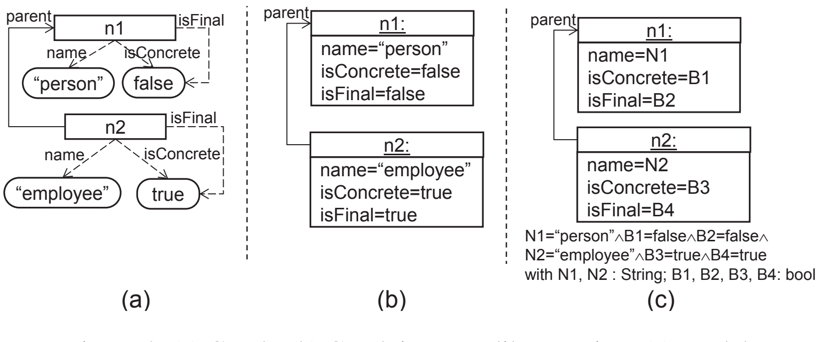


Figure 4: (a) Graph. (b) Graph in UML-like notation. (c) Model.

Similar to symbolic graphs [26], we enrich graphs with formulas over a signature $\Sigma = (S, OP)$ made of a set S of sorts and a set OP of operations [31]. The elements from D are then interpreted as variables (instead of data values) and the set D is partitioned according to the sorts in S (i.e. we consider a S-sorted set). The formulas may use sorted variables from the set D and operations from the set OP. We call such an enriched graph a *model*.

Definition 2 (Model and Grounded Model)**.** *Given a signature $\Sigma = (S, OP)$, a model $M = \langle G; D_S; \alpha\rangle$ is made of a graph $G = \langle V; E; A; D; src_E, tar_E; src_A; tar_A\rangle$, a set of S-sorted variables $D_S = \langle D_s\rangle_{s\in S}$ such that $\biguplus_{s\in S} D_s = D$, and a $\Sigma(D_S)$-formula α with variables from D_S.*

A model M is called grounded *if α is a conjunction of equalities between variables from D_S and constants.*

Remark. The symbol $\biguplus$ denotes disjoint union.

Example 2. Figure 4(c) shows a model equivalent to the graph in Figure 4(b), in the sense that they convey the same information. Instead of having concrete values (e.g., "person" or *false*) as attributes, the model contains variables (e.g., *N1*, *B1*) of some sort. The values the variables can take are given by a formula α. Such formula, and the sort of the variables are depicted at the bottom of the model. The signature contains sorts *String* and *bool*. The model is grounded, because the formula consists of equalities between variables and constants.

For simplicity, we will not consider models having different signatures, but assume a global, common signature Σ for all of them. A graph can be represented as a grounded model by taking as its formula the conjunction of the equalities of each attribute to its value. In such cases, we sometimes prefer using a compact notation that hides the formula and depicts the attribute assignments in their compartments, as in Figure 4(b). In general, formulas may accept different values for the variables they contain. This allows using models to represent patterns (e.g. in the left- and right-hand sides of transformation rules) [26].

Next, we define model morphisms. These are useful to define relations between models, e.g. to indicate an occurrence of the first model into the second one.

Definition 3 (Model morphism)**.** *Given a Σ-algebra $\mathcal{A}$ and two models $M_1 = \langle G_1; D_{S_1}; \alpha_1\rangle$ and $M_2 = \langle G_2; D_{S_2}; \alpha_2\rangle$, a model morphism (short M-morphism), written $m\colon M_1 \to M_2$, is a graph morphism $m\colon G_1 \to G_2$ s.t. $\mathcal{A} \models \alpha_2 \Rightarrow m(\alpha_1)$, where the induced morphism $m_{D_S} : D_{S_1} \to D_{S_2}$ preserves variable sorts.*

Remark. $m(\alpha_1)$ is the formula that results from replacing every variable X in α_1 by the variable $m(X)$. The condition $\mathcal{A} \models \alpha_2 \Rightarrow m(\alpha_1)$ specifies that the algebra satisfies the implication. Hence, every valuation $\mathfrak{f}\colon Var(\alpha_2) \to |\mathcal{A}|$ from the variables in α_2 to the carrier set of the algebra satisfying α_2, can be extended to a valuation $\mathfrak{f}' : D_{S_2} \to |\mathcal{A}|$ satisfying $m(\alpha_1)$.

Example 3. Figure 5 shows an M-morphism example $m\colon M_1 \to M_2$. Model M_1 has two nodes with different values for *isConcrete*, as its equation $X = not\ Y$ indicates. The morphism maps nodes $n2$ and $n3$ in M_1 to nodes with same names in M_2, so that there is an implication $\alpha_2 \Rightarrow m(\alpha_1)$, with $m(X) = B2$ and $m(Y) = B3$, because $N1 = \text{“}Person\text{''} \wedge \cdots \wedge B2 = false \wedge B3 = true \wedge \cdots \wedge F3 = false \Rightarrow B2 = \text{not } B3$. On the contrary, there is no M-morphism if we map the nodes $n3$ and $n2$ in $M1$ to nodes $n2$ and $n1$ in $M2$, respectively, as in this case there is no implication from the formula in $M2$ to the formula in $M1$.

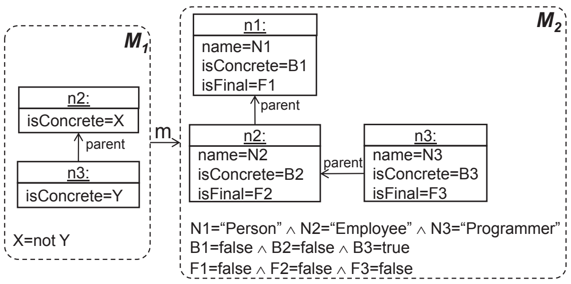


Figure 5: M-morphism example.

We call **Model** the category made of models as objects and M-morphisms as arrows. Similar to [26], it can be shown that **Model** is an adhesive high-level replacement (HLR) category. As shown in [11, 28], these categories provide an adequate framework for several kinds of transformation systems based on the algebraic approach. HLR categories are based on the existence and compatibility of suitable pushouts and pullbacks, which are essential for the so-called van Kampen squares. In such squares, pushouts are stable under pullbacks, and pullbacks are stable under combined pushouts and pullbacks. In our case, the pushout of two models is calculated as the pushout of their underlying graphs and the conjunction of their formulas. Conjunction is the right operation in this case, because given α_1 and α_2 (the formulae of models M_1 and M_2 respectively), the resulting pushout object M_{12} has formula $\alpha_1 \wedge \alpha_2$, and therefore the needed implications $\alpha_1 \wedge \alpha_2 \implies \alpha_1$ and $\alpha_1 \wedge \alpha_2 \implies \alpha_2$ become true. For any other model M_3 s.t. $\alpha_3 \implies \alpha_1$ and $\alpha_3 \implies \alpha_2$, we have $\alpha_3 \implies \alpha_1 \wedge \alpha_2$, the needed condition on the formulae for the existence of a morphism from M_{12} to M_3.

Similarly, the pullback of two models is calculated as the pullback of their underlying graphs and the disjunction of their formulas, existentially quantified on the variables that do not belong to the intersection (see details in [26]). In this way, pushouts are useful to merge models through a common intersection, whereas pullbacks are helpful to calculate the intersection of two models in a common context.

Example 4. The left of Figure 6 shows a pushout example. In particular, model M_{12} is the pushout object of models M_1 and M_2, and is calculated by taking the pushout of the two graphs in M_1 and M_2 and the conjunction of their formulas. The right of the same figure shows a pullback example. The formula in the pullback object K is the disjunction of the formulas in M_1 and M_2, and an existential quantifier is introduced for the variables that are not present in the pullback object, like in the case of variable $B1$.

3.2. Meta-models and concepts

Once we have seen how to represent models, we define meta-models and concepts in a unified way, as they can be represented similarly. Meta-models declare the node types, edge types and attribute types that can appear in models. In order to represent both meta-models and concepts, we use type graphs with inheritance, and therefore the same definition applies to both of them. For simplicity, we omit cardinalities and integrity constraints, which could be given as a set of graph constraints, as in [36].

A meta-model (and a concept) is made of a model M (conforming to Definition 2) to represent the nodes, edges and attributes types. In addition, we distinguish a set Ab of *abstract* nodes.

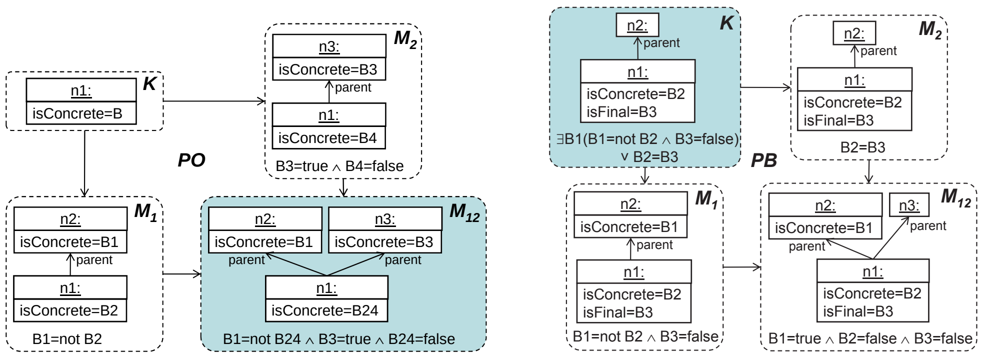


Figure 6: Examples of pushout (left) and pullback (right) for models and M-morphisms.

Finally, the inheritance relation is modelled as a subset of the cartesian product of the nodes, and we define the set *clan*(n) of the (direct and indirect) children of a given node n, including itself. Similar to [21] and [5], we do not explicitly require the inheritance relation to be cycle-free, even though in most practical cases it is.

Definition 4 (Meta-model/Concept)**.** *A meta-model (concept)* $MM = \langle M; Ab; I \subseteq V \times V \rangle$ *is made of a model M, a set* $Ab \subseteq V$ *of abstract nodes and a set I of inheritance relations. V is the set of nodes of model M.*

Given $n \in V$*, its* clan *contains its direct and indirect children, including itself:* $clan(n) = \{n' \in V | (n', n) \in I^*\}$ *with* I^* *the reflexive and transitive closure of I.*

Example 5. Figure 7 shows to the right an example meta-model, where we have used the standard UML notation for class diagrams (inheritance is depicted as edges with hollow arrowhead and the name of abstract classes is in italics). The type T of each attribute is indicated by assigning a variable of sort T to the attribute (see attributes `name` of type *String* and `age` of type *int*). In the following, we will take the convention of showing the sort T next to the attribute name, just like in UML. The formula in meta-models is typically (but not necessarily) equal to $false$, in which case we omit it for simplicity. We could represent integrity constraints, in the style of OCL constraints, by means of graph constraints, but we omit them here for simplicity.

The figure shows to the left the formal definition of the different meta-model elements. We have omitted the formal definition of M, which is straightforward from the picture to its right.

Ab = { Person }
I = { (Employee,Person), (Client,Person), (Manager,Employee) }

clan(Person) = { Person, Employee, Client, Manager }
clan(Employee) = { Employee, Manager }
clan(Manager) = { Manager }
clan(Client) = { Client }

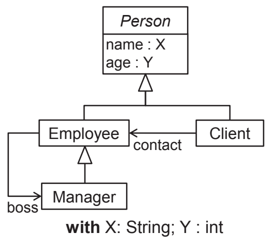


Figure 7: Meta-model example in formal (left) and graphical (right) formats.

The typing of a model with respect to a meta-model will be represented through a special kind of morphism, based on the notion of *clan morphism*, as defined in [5]. A clan morphism from model M_1 to meta-model MM_2 is similar to a model morphism, but it takes into account the semantics of inheritance of the target meta-model. In particular, for each edge e of the graph G_1 (the underlying graph of M_1) that is mapped to an edge e' of G_2 (the underlying graph of MM_2), we allow the source node of e to be mapped to any node in the clan of the source node of e'. This means that e' can be defined in an ancestor of the node to which the source of e is mapped. Formally, $f_V(src^1_E(e)) \in clan(src^2_E(f_E(e)))$, and similar for the target of edges. In addition, we allow mapping an attribute of a node to an attribute of a supertype the node is mapped into. Formally, $f_A(src^1_A(a)) \in clan(src^2_A(f_A(a)))$.

Definition 5 (Clan morphism)**.** *Given a model $M_1 = \langle G_1; D_{S_1}; \alpha_1\rangle$, a meta-model $MM_2 = \langle M_2 = \langle G_2; D_{S_2}; \alpha_2\rangle; Ab_2; I_2\rangle$, and an algebra $\mathcal{A}$, a clan morphism $f\colon M_1 \to MM_2$ is a collection of mappings ($f_V\colon V_1 \to V_2, f_E\colon E_1 \to E_2, f_A\colon A_1 \to A_2, f_D\colon D_1 \to D_2$) s.t.:*

1. $\forall e_1 \in E_1$: $f_V(src^1_E(e_1)) \in clan(src^2_E(f_E(e_1))) \wedge f_V(tar^1_E(e_1)) \in clan(tar^2_E(f_E(e_1)))$,
2. $\forall a \in A_1$: $f_A(src^1_A(a)) \in clan(src^2_A(f_A(a))) \wedge f_A(tar^1_A(a)) = tar^2_A(f_A(a))$,

and with $\mathcal{A} \models \alpha_2 \Rightarrow m(\alpha_1)$, where the induced morphism $f_{D_S}\colon D_{S_1} \to D_{S_2}$ preserves variable sorts.

Remark. A clan-morphism from a model M to a meta-model MM that does not have inheritance relations, is equivalent to an M-morphism, because in such a case, the clan of any element contains only such element, and conditions 1 and 2 reduce to the standard commutativity conditions required from graph morphisms [11]. Please note that, at this point, we are not concerned yet with the abstractness of nodes.

Example 6. Figure 8 shows a clan-morphism between a model M and a meta-model MM. The morphism is valid because, for example, node c is mapped to *Client*, and the *name* attribute of c is mapped to *name*, defined on *Person*, however *Client* $\in$ *clan*(*Person*).

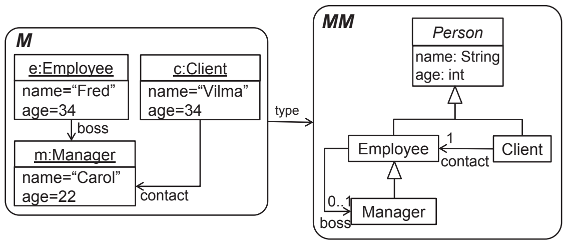


Figure 8: A clan-morphism.

Next, we define morphisms between meta-models, called MM-morphisms, also based on the notion of clan morphism. Basically, an MM-morphism $f\colon MM_1 \to MM_2$ is a clan morphism $M_1 \to MM_2$ from M_1, the underlying model of MM_1, to MM_2, subject to some extra conditions. Our MM-morphisms follow the same conditions as I-Graph morphisms in [21], but MM-morphisms consider attributes and abstract nodes.

Definition 6 (MM-morphism)**.** *Given two meta-models $MM_i = \langle M_i; Ab_i; \alpha_i \rangle$ (for $i = 1, 2$) and an algebra $\mathcal{A}$, a meta-model morphism (short MM-morphism) from MM_1 to MM_2, written $f\colon MM_1 \to MM_2$, is a clan morphism $f\colon M_1 \to MM_2$ from the underlying model of MM_1 to MM_2 that preserves the inheritance hierarchy (if $(u, v) \in I_{MM_1}$, then $(f(u), f(v)) \in I^*_{MM_2}$), and the* abstractness *of nodes ($u \in A \Leftrightarrow f(u) \in A'$).*

Example 7. Figure 9 shows an MM-morphism example. *Client* in MM_1 is mapped to *Client* in MM_2, and their *age* attributes are mapped as well. This is possible because *Client* in MM_2 inherits attribute *age* from *Person*, and hence, $f_A(src^1_A(age)) = Client \in clan(src^2_A(f_A(age)) = Person) = \{Person, Employee, Client, Manager\}$. Similarly, *Contact* is mapped to *Manager*, and edge *contact* in MM_1 is mapped to *contact* in MM_2. This can be done because $f_V(tar^1_E(contact)) = Manager \in clan(tar^2_E(f_E(contact)) = Employee) = \{Employee, Manager\}$.

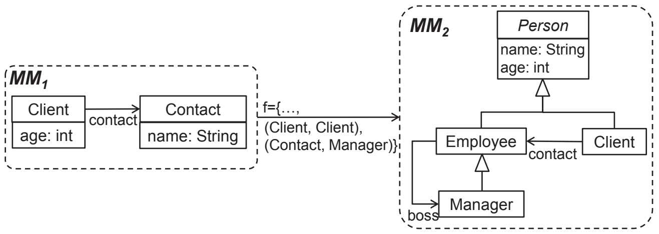


Figure 9: Meta-model morphism example.

Meta-models and MM-morphisms form the category **MetaModel**. It is interesting to note that a model M can be represented as a meta-model $F(M) = \langle M; \emptyset; \emptyset \rangle$, with empty set of abstract nodes and empty inheritance relation. Similarly, an M-morphism $f\colon M_1 \to M_2$ can be represented by an MM-morphism $F(f)\colon \langle M_1; \emptyset; \emptyset \rangle \to \langle M_2; \emptyset; \emptyset \rangle$. This is so as, in absence of inheritance, clan-morphisms are equivalent to M-morphisms. Formally, this means that there is a free functor $F\colon$ **Model** $\to$ **MetaModel** from the category **Model** of models and M-morphisms, to the category **MetaModel** of meta-models and MM-morphisms. In addition, the forgetful functor $U\colon$ **MetaModel** $\to$ **Model** maps meta-models to models by simply neglecting the abstract nodes and inheritance relation. The advantage of this fact is that models and meta-models can be treated uniformly.

Due to the existence of the free functor F, next we define the *conformance* relation between a model M and a meta-model MM as an MM-morphism $type\colon F(M) \to MM$.

Definition 7 (Conformance)**.** *A model M conforms to a meta-model MM if $\exists type\colon \langle M; \emptyset; \emptyset \rangle \to MM$.*

Example 8. Figure 8 shows that M conforms to MM because there is a morphism $type\colon \langle M; \emptyset; \emptyset \rangle \to MM$

Remark. In the following, abusing of notation and when no confusion arises, we just write $type\colon M \to MM$ instead of $type\colon \langle M; \emptyset; \emptyset \rangle \to MM$.

MM-morphisms have to preserve the abstractness of nodes. Hence, the existence of a morphism $type\colon M \to MM$ implies that no node in M can be mapped to an abstract node in MM. MM-morphisms can be composed, and hence, we can retype a model. If we have $type\colon M \to MM_1$ and $f\colon MM_1 \to MM_2$, then we have a typing $f \circ type\colon M \to MM_2$.

Similar to S-reflecting I-morphisms in [21], we next introduce a restricted kind of MM-morphism (that we call SMM-morphism) that reflect the subtyping relations in the source meta-model. This kind of morphisms are needed in order to build pushouts and pullbacks componentwise, as demonstrated in [21].

Definition 8 (SMM-morphism)**.** *An SMM-morphism* $f\colon MM_1 \to MM_2$ *is an injective MM-morphism that reflects subtyping:* $\forall (c_2, a_2) \in I^*_{MM_2}, c_1 \in V_1$*:* $c_2 = f(c_1) \implies \exists a_1 \in V_1 : f(a_1) = a_2 \wedge (c_1, a_1) \in I^*_{MM_1}$*, where* V_1 *is the set of nodes of* MM_1*.*

Example 9. The MM-morphism in Figure 9 is not an SMM-morphism. This is so as we have $(Client, Person) \in I_{MM_2}$ and $f(Client) = Client$. However, there is no node a_1 in MM_1 such that $f(a_1) = Person$. Similarly, $(Manager, Employee) \in I_{MM_2}$ and $f(Contact) = Manager$. However, there is no node a_1 in MM_1 such that $f(a_1) = Employee$. Hence, SMM-morphisms require the inheritance relations in the co-domain meta-model to be reflected in the source. Figure 10 shows an SMM-morphism example. In this case, the morphism is an SMM-morphism because the inheritance relation in the target is reflected in the source (this time, *Contact* is mapped to *Employee*). Please note that the target meta-model may contain unmapped subtypes (like *Manager*) of mapped elements (like *Employee*), but not unmapped supertypes.

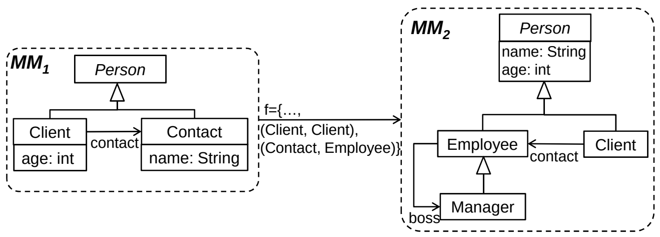


Figure 10: An SMM-morphism.

Similar to [21], pushouts (POs) and pullbacks (PBs) of meta-models along SMM-morphisms can be built componentwise. Pushouts (POs) of meta-models are defined by making the PO of the underlying models and taking the union of their sets of abstract nodes and inheritance relations. Pullbacks (PBs) are defined by the PB of the underlying models and the intersection of their abstract and inheritance sets.

Example 10. Figure 11 shows an example of pushout of two meta-models MM_1 and MM_2 through a common part K, which yields the pushout object M_{12}. In this case, g is not an SMM-morphism (because *Contact* is mapped to *Manager*), but f is an SMM-morphism, and so the pushout object MM_{12} exists an can be built componentwise. Moreover, f^* becomes an SMM-morphism as well.

In order to better understand the need for SMM-morphisms, Figure 12 shows a PO attempt along two MM-morphisms, f and g, none of which is an SMM-morphism. In this case, the problem is that, if we just take the union of the inheritance relation, we are unable to place appropriatelly the *marriedto* reference.

3.3. Meta-model closures

We now define an **inheritance closure** for meta-models, which we will use in Section 4 to define the binding from concepts to meta-models. This operation removes a set $I' \subseteq I$ of

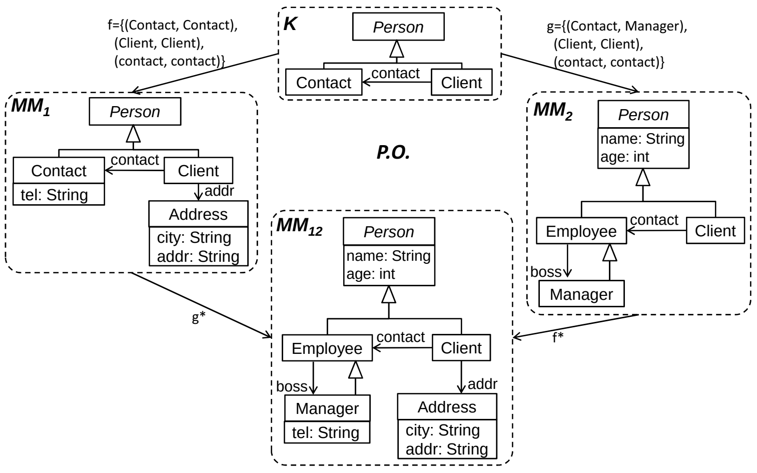


Figure 11: Example of pushout of meta-models.

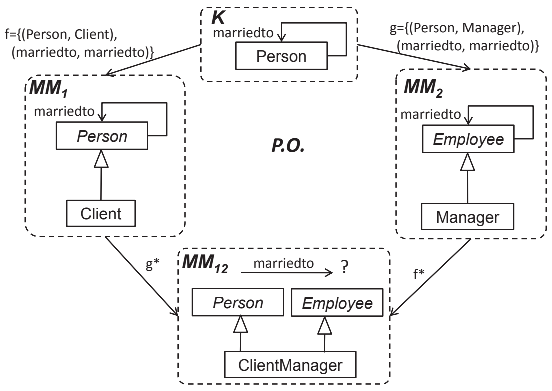


Figure 12: Attempting a pushout along non SMM-morphisms.

inheritance tuples, making explicit the semantics of inheritance: it replicates in the children nodes the attributes and edges starting/ending in the parent nodes. This closure can only be performed if the inheritance relations in I' are not continued outside I', as otherwise, the attributes and relations of the ancestors nodes outside I' would not get copied to the children nodes.

Definition 9 (Inheritance closure)**.** *Let $MM = \langle M; Ab; I\rangle$ be a meta-model and $I' \subseteq I$ a set of tuples s.t. $(p, a) \in I \setminus I' \implies \nexists (c, p) \in I'$. We define the closure of MM with respect to I', written $\overline{MM}_{I'} = \langle M'; Ab; I \setminus I'\rangle$, with $M' = \langle\langle V; E'; A'; D; src'_E, tar'_E; src'_A; tar'_A\rangle; D_S; \alpha\rangle$, as follows:*

- *$E' = E \cup \{(c, e) \mid e \in E \wedge (c, p) \in I'^+ \wedge c \neq p \wedge src_E(e) = p\} \cup \{(e, c) \mid e \in E \wedge (c, p) \in I'^+ \wedge c \neq p \wedge tar_E(e) = p\}$ (edges from/to p are replicated in children nodes).*
- *$A' = A \cup \{(c, a) \mid a \in A \wedge (c, p) \in I'^+ \wedge c \neq p \wedge src_A(a) = p\}$ (attributes in p are replicated).*
- *The source (src'_E) and target (tar'_E) functions of edges is defined as:*

  - $\forall e \in E, src'_E(e) = src_E(e), tar'_E(e) = tar_E(e)$ *(source/target of original edges is preserved).*
  - $\forall (c,e) \in E' \setminus E, src'_E((c,e)) = c, tar'_E((c,e)) = tar_E(e)$ *(replicas of edges from p, are moved to c).*
  - $\forall (e,c) \in E' \setminus E, src'_E((e,c)) = src_E(e), tar'_E((e,c)) = c$ *(replicas of edges to p, are directed to c).*

- *The source (src'_A) and target (tar'_A) of attributes is defined as:*
  - $\forall a \in A,\ src'_A(a) = src_A(a), tar'_A(a) = tar_A(a)$ *(owner of original attributes is preserved).*
  - $\forall (c,a) \in A' \setminus A, src'_A((c,a)) = c; tar'_A((c,a)) = tar_A(a)$ *(replicas of attributes in p, are assigned to c).*

where I'^+ is the transitive closure of relation I'.

Remark. We have used tuples of the form (c,e) for replicas of edges starting in p, (e,c) for replicas of edges ending in p, and (c,a) for replicas of attributes belonging to p.
Remark. The resulting meta-model is well formed, because we only remove inheritance relations. The closure makes explicit the semantics of such inheritance relations by replicating in c all attributes and edges starting and ending in the parent p. If we only remove one inheritance tuple (c,p), we simply write $\overline{MM}_{(c,p)}$ for such closure, while for the closure of the whole I, we write $\overline{MM}$. Please note that this procedure is a slight generalization of the closure operation presented in [5], which considers all inheritance tuples, while here we consider subsets.

Example 11. Figure 13 shows a meta-model MM and the different steps in applying the closure for each of the inheritance relations until its inheritance closure $\overline{MM}$ is obtained. At each step, the set I' is made of just one tuple. In $\overline{MM}$, the attributes of *Person* have been replicated in all its subclasses, and relations *contact* and *boss* get replicated as well when the inheritance relation between *Manager* and *Employee* is eliminated. In case of naming conflicts when replicating edges, we take the convention of assigning to the new edges the name of the original edge (e.g. *boss*) followed by the target node name (e.g. *Manager*, yielding the edge name *bossManager*). The figure also shows $\overline{MM}^{Person}$, the concrete closure, which we explain next.

The **concrete closure** of a meta-model MM consists in removing a set $Ab' \subseteq Ab$ of abstract classes (with none of the classes in Ab' having children) and all its owned attributes and adjacent edges.

Definition 10 (Concrete closure). *Let $MM = \langle M; Ab; I\rangle$ be a meta-model, and $Ab' \subseteq Ab$ a set of abstract classes s.t. $\forall a \in Ab'\ \nexists (p,a) \in I$. We define the concrete closure $\overline{MM}^{Ab'} = \langle M'; Ab \setminus Ab'; I\rangle$, with $M' = \langle\langle V \setminus Ab'; E'; A'; D; src'_E, tar'_E;\ src'_A; tar'_A\rangle; D_S; \alpha\rangle$ as follows:*

- $E' = E \setminus \{\, e \mid src_E(e) \in Ab' \vee tar_E(e) \in Ab'\}$.
- $A' = A \setminus \{\, at \mid src_A(at) \in Ab'\}$.
- *src'_E and tar'_E are equal to src_E and tar_E, restricted to E'.*
- *src'_A and tar'_A are equal to src_A and tar_A, restricted to A'.*

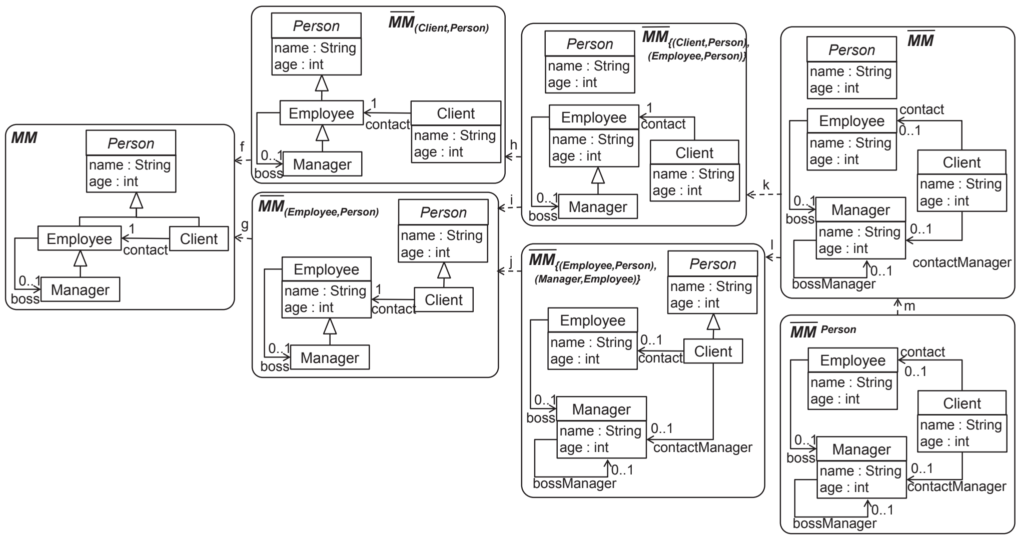


Figure 13: Inheritance and concrete closures of a meta-model MM.

Remark. The resulting meta-model $\overline{MM}^{Ab'}$ is well formed because we eliminate all edges starting or ending in the classes in Ab', as well as its attributes. Intuitively, the semantics is preserved, because each class $a \in Ab'$ has no children, and being abstract, it can not be instantiated in instance models. Following the previous notation, if $Ab' = \{a\}$, we write $\overline{MM}^{a}$.

Example 12. Figure 13 shows the concrete closure $\overline{MM}^{Person}$ of $\overline{MM}$, resulting from removing the abstract node *Person*. While the figure makes the concrete closure as a last step, it could be done in each model where *Person* has no children.

As in [5], there is a universal morphism $u\colon \overline{MM}_{I'}^{Ab'} \to MM$ from any closure $\overline{MM}_{I'}^{Ab'}$ of a meta-model MM to MM. We obtain a lattice with $\overline{MM}_{I}^{Ab}$ at the bottom and MM at the top, because there is a morphism $u\colon \overline{MM}_{I'}^{Ab'} \to \overline{MM}_{I''}^{Ab''}$ if $Ab'' \subseteq Ab'$ and $I'' \subseteq I'$. For instance, in Figure 13, there is a universal morphism $u = g \circ i \circ k \circ m\colon \overline{MM}^{Person} \to MM$, identifying the replicated edges and the original ones (e.g., $u(boss) \mapsto boss$ and $u(bossManager) \mapsto boss$).

As in [5], any model M typed by MM is typed by any closure $\overline{MM}_{I'}^{Ab'}$, and vice versa.

4. Algebraic Adapters and Bindings

In [6], we defined bindings as injective MM-morphisms from concepts to meta-models. This is possible because we represent both concepts and meta-models uniformly, as given by Definition 4. However, this is not enough if we want to allow structural heterogeneities between them, like those in Figure 2, for which we need to define mappings at the model level as well.

Roughly, a binding is made of an MM-morphism, called *correspondence*, from a subset $\overline{C'}$ of a suitable closure of the concept to the meta-model. $\overline{C'}$ contains all classes, fields and edges that can be directly mapped to the meta-model. Thus, the correspondence declares injective mappings from each class in $\overline{C'}$ to some class in the meta-model, and from the fields and edges in $\overline{C'}$ to some field and edge in the meta-model. Choosing a closure of the concept C allows leaving unmapped some abstract class in C, which gets deleted by the chosen concrete closure, but the fields and edges inherited by its children classes need to be mapped, as made explicit by the closure. To deal with the heterogeneities, the correspondence is complemented with *adapters* at the model level, as we will see in Definition 16.

Definition 11 (Correspondence)**.** *Given a concept $C = \langle M; Ab; I \rangle$ and a meta-model MM, a correspondence is an injective MM-morphism $m \colon \overline{C'} \to MM$, with $\overline{C'} \hookrightarrow \overline{C}_{I'}^{Ab'}$ for some $Ab' \subseteq Ab$ and $I' \subseteq I$, where $\overline{C'}$ and $\overline{C}_{I'}^{Ab'}$ have the same graph nodes ($V_{\overline{C'}} = V_{\overline{C}_{I'}^{Ab'}}$) and the same set of inheritance relations.*

Remark. $\overline{C'}$ contains the elements that can be directly mapped to elements in MM. The elements in $\overline{C}_{I'}^{Ab'} \setminus \overline{C'}$ need to be mapped at the model level through adapters, as we will show in Definition 16. As a difference from [6], we do not need to map abstract classes in the concept, but we could. We use some closure of the concept because, in case we do not map an abstract class $a \in Ab'$, we need to map a's replicated fields and edges in the closure $\overline{C}_{I'}^{Ab'}$. Moreover, we need to map all non-abstract nodes in $\overline{C}$, because of the condition $V_{\overline{C'}} = V_{\overline{C}_{I'}^{Ab'}}$.

Remark. $\overline{C'} \hookrightarrow \overline{C}_{I'}^{Ab'}$ is an SMM-morphism, because it is injective, and reflects the inheritance relation, because $\overline{C'}$ has the same inheritance relations as $\overline{C}_{I'}^{Ab'}$.

Example 13. In order to illustrate correspondences, we retake the example introduced in Section 2, Figure 1. Figure 14 shows, from right to left, a concept for object oriented notations, a concrete closure $\overline{Concept}^{NamedElement}$ of this concept, and a correspondence m from a subgraph $\overline{Concept'}$ of the closure to a simplified UML2 meta-model. The mapped elements in the UML2 meta-model are highlighted in gray. The chosen closure ($\overline{Concept}^{NamedElement}$) has eliminated the inheritance relations ending at *NamedElement*, replicating the attribute *name* in the children *Node* and *Field*, and then has removed the class itself. We have chosen this closure as the simplified UML2 meta-model does not define a suitable class to bind *NamedElement*, and therefore we need to map its *name* field on every subclass. The *inc* morphism is an SMM-morphism.

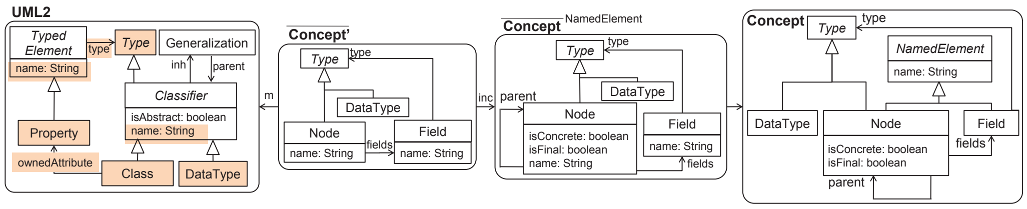


Figure 14: Correspondence example.

In the example, the concept elements without a direct mapping to meta-model elements, like association *parent* or the attributes *isConcrete* and *isFinal*, are not in $\overline{Concept'}$. Instead, we define *adapters* showing how to resolve these structural heterogeneities. Adapters are co-span of M-morphisms having concept and meta-model instances (objects of category **Model**) at both ends, and a gluing model (called *store*) relating their elements. The meta-model of the *store* is the gluing of the chosen concept closure and the meta-model, as shown in the Definition 12.

$$\begin{tikzcd}
\overline{C'} \arrow[r, "m"] \arrow[d, hook] \arrow[dr, phantom, "P.O."] & MM \arrow[d, "f"] \\
\overline{C}^{Ab'}_{I'} \arrow[r, "g"] & Store
\end{tikzcd}$$

Figure 15: Store meta-model.

Intuitively, the *store* contains the new elements that need to be added to the concrete meta-model, so that it becomes a *subtype* of the concept. The idea is that those extra elements are to be thought as *derived* (check Figure 3) so that the GT template can work on instances of the store. The way these derived elements are calculated is given by the adapters.

Definition 12 (Store meta-model)**.** *Given a correspondence* $m\colon \overline{C'} \to MM$*, with* $\overline{C'} \hookrightarrow \overline{C}^{Ab'}_{I'}$*, the store meta-model is given by the pushout object of* $MM \xleftarrow{m} \overline{C'} \hookrightarrow \overline{C}^{Ab'}_{I'}$ *(see Figure 15).*

Remark. Such pushout exists because $\overline{C'} \hookrightarrow \overline{C}^{Ab'}_{I'}$ is an SMM-morphism.

Example 14. Figure 16 shows the *Store* meta-model for our example, which is the result of merging $\overline{Concept}^{NamedElement}$ and the meta-model of UML2 through $\overline{Concept'}$. Thus, it contains once the elements from $\overline{Concept}^{NamedElement}$ and UML2 identified by the two morphisms (their names are concatenated, like *NodeClass*), and includes the elements from the concept and the meta-model that are not identified. The *Store* extends *UML2* with derived information (c.f. Figure 3).

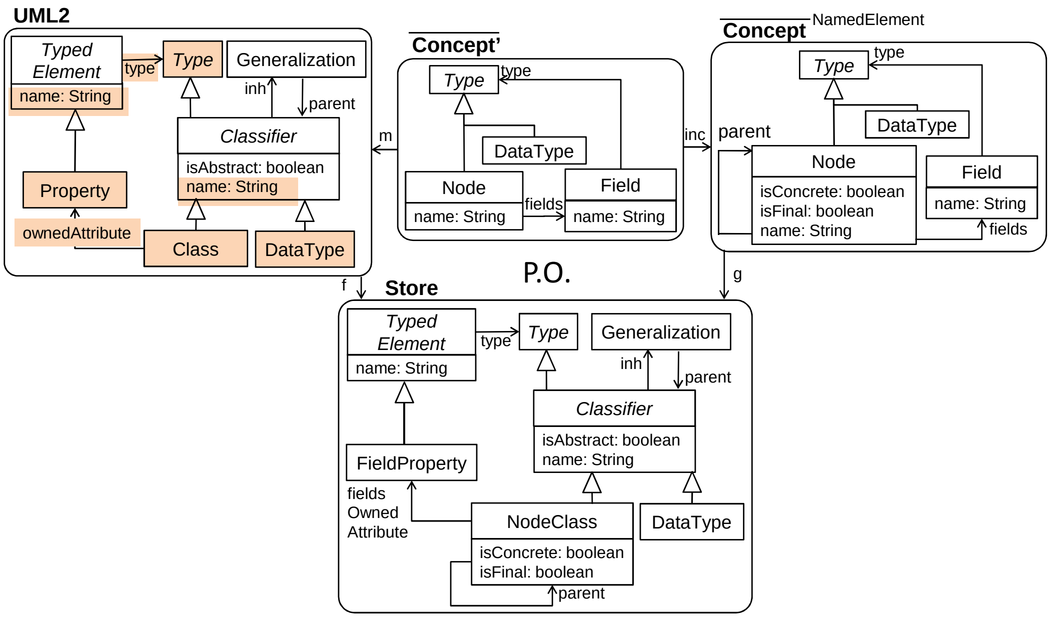


Figure 16: Store meta-model for the running example.

We use adapters to specify heterogeneities between concepts and meta-models. Adapters are co-spans typed by $\overline{C}_{I'}^{Ab'} \xrightarrow{g} Store \xleftarrow{f} MM$. The two M-morphisms in the co-span should be jointly surjective, so that each element in the *store* model is mapped from some element in the models to the left or right (i.e. the store does not introduce new elements). Moreover, adapters must preserve the compatibility of the formulas at both ends of the co-span and the store. The role of the formula in the store is to relate variables in the concept and meta-model instances. Thus, this formula should contain all terms from both ends in the co-span, and maybe other terms relating variables from both ends, but never constraining only one end. As we will see later, this condition is required to obtain rules with equivalent applicability.

Definition 13 (Adapter)**.** *An adapter is a jointly surjective co-span of injective M-morphisms with the form:*

$$CI = \langle G_{CI}; D_S^{CI}; \alpha_{CI} \rangle \xrightarrow{c} S = \langle G_S; D_S^S; \alpha_S \rangle \xleftarrow{m} M = \langle G_M; D_S^M; \alpha_M \rangle$$

where α_S is a satisfiable constraint of the form $c(\alpha_{CI}) \wedge m(\alpha_M) \wedge \alpha'$, and α' does not contain terms with variables in $c(D_S^{CI})$ or in $m(D_S^M)$ only.

Example 15. Figure 17 shows some adapters for our running example. While adapter (b) indicates that *isConcrete* and *isAbstract* have opposite meanings, (a) identifies the value *true* for *isFinal* with the marker class *Final*, (d) assigns a default value *false* to *isFinal*, and (c) identifies the relation *parents* in the concept to an intermediate object of type *Generalization* in the UML2 meta-model that realizes the same notion. We also require adapters for the elements mapped by the correspondence, like (e), which can be automatically derived as Definition 14 will show.

Regarding the formulas, adapter (b) has the formula $\alpha_S \equiv X = not\ Y$. This conforms to Definition 13, as $\alpha_{CI} = true$, $\alpha_M = true$, and $\alpha_S \equiv true \wedge true \wedge X = not\ Y \equiv X = not\ Y$. Adding the term $X = false$ to the formula would not be allowed, as this term only constrains the left model but the left model does not include the term.

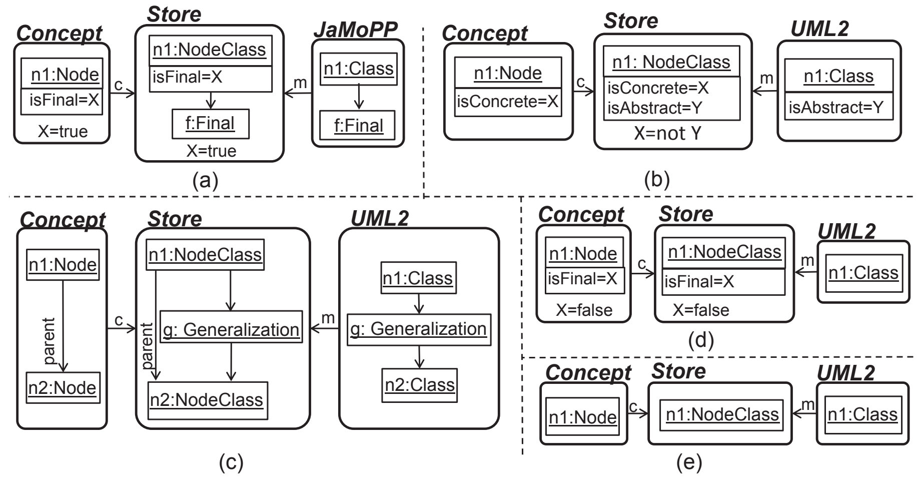


Figure 17: Some adapters: (a) field to class, (b) field value adapter, (c) relation to intermediate class, (d) field to default value, (e) class mapping.

Next, we show how to generate adapters (as the one in Figure 17(e)) from a correspondence m. These trivially bridge concept and meta-model elements, as given by the direct mapping $m\colon \overline{C'} \to MM$.

Definition 14 (Generated Adapter)**.** *Given a correspondence $m\colon \overline{C'} \to MM$, the set $GEN(m) = NODES(m) \cup EDGES(m) \cup ATTRS(m)$ of generated adapters, typed over $\overline{C}^{Ab'}_{I'} \xrightarrow{g} Store \xleftarrow{f} MM$ is built as follows:*

- *$\forall n \in V_{\overline{C'}}$: $CI_n \xrightarrow{c_n} S_n \xleftarrow{m_n} M_n \in NODES(m)$, with each CI_n, S_n and M_n made of a single node, typed over n, $g(n)$ and $m(n)$, respectively.*
- *$\forall e \in E_{\overline{C'}}$: $CI_e \xrightarrow{c_e} S_e \xleftarrow{m_e} M_e \in EDGES(m)$, with each CI_e, S_e and M_e made of a single edge (with corresponding source/target nodes), typed over e, $g(e)$ and $m(e)$, respectively.*
- *$\forall a \in A_{\overline{C'}}$: $CI_a \xrightarrow{c_a} S_a \xleftarrow{m_a} M_a \in ATTRS(m)$, with each CI_a, S_a and M_a made of a single attribute (with corresponding source/target nodes), typed over a, $g(a)$ and $m(a)$, respectively.*

Example 16. Figure 18 shows the generated adapters from the correspondence $m\colon \overline{Concept'} \to \overline{UML2}$ of Figure 16. Each adapter in (a) is generated for each one of the nodes in $\overline{Concept'}$ (*Node*, *Field*, *DataType* and *Type*) in Figure 16, each adapter in (b) is generated for each one of the edges in $\overline{Concept'}$ (*type* and *fields*), while the adapters in (c) are generated for the attributes in $\overline{Concept'}$ (attributes *name* in *Node* and *Field*).

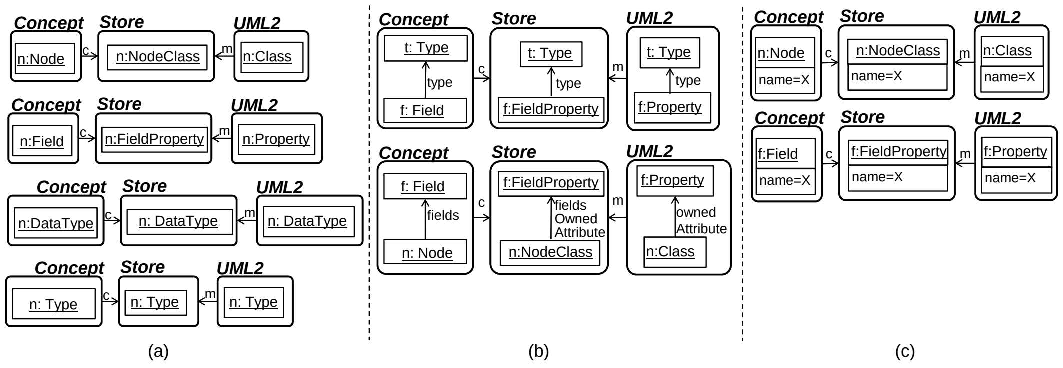


Figure 18: Generated adapters: (a) $NODES(m)$, (b) $EDGES(m)$, (c) $ATTRS(m)$.

Next, we define bindings from concepts to meta-models. A binding is made of a correspondence and a set of store-compatible adapters. Compatibility of adapters asks for coherence of the inclusions between the adapters. We first define the desired properties of adapters and then define the binding.

Definition 15 (Compatible adapters)**.** *Let P be a set of adapters, and A_i, A_j, two adapters of the set, $\{A_i = \langle CI_i \xrightarrow{c_i} S_i \xleftarrow{m_i} M_i\rangle, A_j = \langle CI_j \xrightarrow{c_j} S_j \xleftarrow{m_j} M_j\rangle\} \subseteq P$.*

We say that A_i is store-compatible with A_j, written $A_i \leftrightsquigarrow A_j$, if $\forall s_{ij}\colon S_i \to S_j$, $\exists m_{ij}\colon M_i \to M_j$, $\exists c_{ij}\colon CI_i \to CI_j$, (all injective) s.t. (1) and (2) in Figure 19 are pullbacks. Two adapters A_i and A_j are store-compatible if $A_i \leftrightsquigarrow A_j$ and $A_j \leftrightsquigarrow A_i$.

We say that A_i is forward-compatible with A_j, written $A_i \rightsquigarrow A_j$, if $A_i \leftrightsquigarrow A_j$ and $\forall c_{ij}\colon CI_i \to CI_j, \exists s_{ij}\colon S_i \to S_j$, $\exists m_{ij}\colon M_i \to M_j$, (all injective) s.t. (1) and (2) in Figure 19 are pullbacks. Two adapters A_i and A_j are forward-compatible if $A_i \rightsquigarrow A_j$ and $A_j \rightsquigarrow A_i$.

Similarly, A_i is backwards-compatible with A_j, written $A_i \leftsquigarrow A_j$, if $A_i \leftrightsquigarrow A_j$ and $\forall m_{ij}\colon M_i \to M_j, \exists s_{ij}\colon S_i \to S_j$, $\exists c_{ij}\colon CI_i \to CI_j$, (all injective) s.t. (1) and (2) in Figure 19 are pullbacks. Two adapters A_i and A_j are backwards-compatible if $A_i \leftsquigarrow A_j$ and $A_j \leftsquigarrow A_i$.

Remark. While every binding needs to be store-compatible, we demand forward-compatibility when amalgamating adapters given an instance of the concept, because we need to translate instances of the concept into instances of the meta-model. Typically, this happens when we need to create meta-model rules, given rules defined over the concept. We require backwards-compatibility when amalgamating the adapters given an instance of the meta-model (with the purpose of creating an instance of the concept).

Remark. The adapters in $GEN(m)$ are store-, backwards- and forward-compatible. First, each subset $NODES(m)$, $EDGES(m)$ and $ATTRS(m)$ is store-compatible, because by construction there is no $s_{ij}\colon S_i \to S_j$ among adapters in the same subset. This is so as the correspondence is injective, and hence both $g\colon \overline{C}^{Ab'}_{I'} \to Store$ and $f\colon MM \to Store$ are injective, which means that not two elements in $\overline{C}^{Ab'}_{I'}$ (or MM) can be mapped to the same element in $Store$. Second, by construction there is no $c_{ij}\colon CI_i \to CI_j$ among adapters in the same subset. Then, for each adapter $CI_n \overset{c_n}{\to} S_n \overset{m_n}{\leftarrow} M_n \in NODES(m)$ s.t. $\exists CI_a \overset{c_a}{\to} S_a \overset{m_a}{\leftarrow} M_a \in ATTRS(m)$, and commuting $CI_n \to CI_a$, $S_n \to S_a$, $M_n \to M_a$ we have that the required pullbacks exist, because $CI_a \overset{c_a}{\to} S_a \overset{m_a}{\leftarrow} M_a$ contains a copy of $CI_n \overset{c_n}{\to} S_n \overset{m_n}{\leftarrow} M_n$ (see Figure 18). A similar reasoning follows for each adapter $CI_n \overset{c_n}{\to} S_n \overset{m_n}{\leftarrow} M_n \in NODES(m)$ s.t. $\exists CI_e \overset{c_e}{\to} S_e \overset{m_e}{\leftarrow} M_e \in EDGES(m)$ and commuting morphisms. The same reasoning holds for backwards-compatibility.

$$\begin{tikzcd}
CI_i \arrow[r, "c_i"] \arrow[d, "c_{ij}"'] \arrow[dr, phantom, "(1)"] & S_i \arrow[d, "s_{ij}"] \arrow[dr, phantom, "(2)"] & M_i \arrow[l, "m_i"'] \arrow[d, "m_{ij}"] \\
CI_j \arrow[r, "c_j"] & S_j & M_j \arrow[l, "m_j"']
\end{tikzcd}$$

Figure 19: Adapters compatibility.

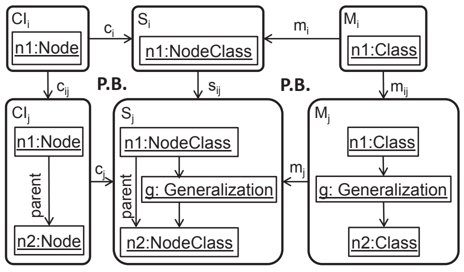


Figure 20: Example of adapters compatibility.

Example 17. Figure 20 shows two forward and backwards compatible adapters. The one on the top belongs to $GEN(m)$, while the second one has been manually specified to resolve an heterogeneity between the concept and the meta-model. There are two M-morphisms from S_i to S_j, and in both cases, corresponding morphisms $c_{ij}\colon CI_i \to CI_j$ and $m_{ij}\colon M_i \to M_j$ making the squares pullbacks. Similarly, there are two M-morphisms from CI_i to CI_j, and in both cases there are corresponding M-morphisms from S_i and M_i such that the squares are pullbacks. Finally, there are also two M-morphisms from M_i to M_j and corresponding M-morphisms from S_i and CI_i making the squares pullbacks.

Looking at the adapters (d) and (e) in Figure 17 (that we denote A_d and A_e), we see that A_e is forward-compatible with A_d, and A_d is (vacuously) forward-compatible with A_e. However, while A_e is backwards-compatible with A_d, A_d is not backward compatible with A_e.

Now, we are ready to define a binding, as a correspondence and a set of store-compatible adapters. At this moment, we do not require either forward- or backward-compatibility, which will be required later depending on the usage of the binding (adaptation of rules, or adaptation of models).

Definition 16 (Binding). *A binding from a concept C to a meta-model MM is a tuple*

$$B_{C,MM} = \langle m \colon \overline{C'} \rightarrow MM, P = \{CI_i \xrightarrow{c_i} S_i \xleftarrow{m_i} M_i\}\rangle$$

where m is a correspondence, and P is a set of store-compatible adapters, with each CI_i typed on $\overline{C}_{I'}^{Ab'}$, S_i typed on the $Store$, and M_i typed on MM, and s.t. $GEN(m) \subseteq P$.

In order to be useful, bindings need to be complete with respect to the concept C. This means that every node, edge and possible attribute value in the concept should be considered by the binding. Please note that, given a correspondence m, using the set $GEN(m)$ as the set of adapters in the binding is normally not enough for completeness[1], because those adapters only cover the elements in $\overline{C'}$, which is included in a closure $\overline{C}_{I'}^{Ab'}$ of the concept C. This means that we should manually provide adapters for the elements in $\overline{C}_{I'}^{Ab'}$ but not in $\overline{C'}$.

Completeness is necessary because we will use adapters for transforming rules conforming to the concept into rules conforming to the meta-model, as well as to translate instances of the meta-model into instances of the concept.

Definition 17 (Complete binding). *A binding $B_{C,MM} = \langle m, P\rangle$ is complete with respect to C if:*

1. *for each node N in $\overline{C}_{I'}^{Ab'}$, $\exists CI \xrightarrow{c} S \xleftarrow{m} M \in P$ with only one instance node n in CI s.t. $type(n) = N$.*

2. *given each E in $\overline{C}_{I'}^{Ab'}$, let $Adapt_E = \{CI_i \xrightarrow{c_i} S_i \xleftarrow{m_i} M_i \in P \mid CI_i$ has exactly one edge e_i with $type(e_i) = E$ adjacent to at most two nodes$\}_{i \in I} \subseteq P$. Such set should cover the clan of the source and target node types of E:*
   $\forall N_S \in clan(src(E)), \forall N_T \in clan(tar(E))$:
   $[\exists CI_i \xrightarrow{c_i} S_i \xleftarrow{m_i} M_i \in Adapt_E \mid e_i \in CI_i \wedge N_S \in clan(type(src(e_i))) \wedge N_T \in clan(type(tar(e_i)))]$.

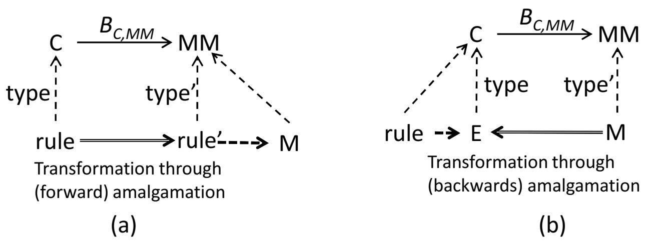
3. *for each field A in $\overline{C}_{I'}^{Ab'}$, let $Adapt_A = \{CI_j \xrightarrow{c_j} S_j \xleftarrow{m_j} M_j \mid CI_j$ has exactly one instance node n_j and exactly one attribute a_j s.t. $type(a_j) = A\}_{j \in J} \subseteq P$. Such set should cover all possible valuations for a for each node type $N \in clan(src(A))$ as follows:*
   $\forall N \in clan(src(A))$: $[\bigvee_{N \in clan(type(src(a_j)))} \alpha_{CI_j}$ *covers all possible valuations for* $a]$.

Remark. $GEN(m)$ is complete w.r.t. $\overline{C}'$. Hence, $P \setminus GEN(m)$ should properly cover $\overline{C}_{I'}^{Ab'} \setminus \overline{C}'$.
Remark. Condition 2 ensures that there is an adapter for each edge type E, covering all combinations of subclasses of the source and target nodes of E. One adapter having a node n covers all

[1]unless there are no heterogeneities between the concept and the meta-model, so that the correspondence alone is enough.

subclasses of $type(n)$, and hence one adapter for E is enough if it has an edge e_i with $type(e_i) = E$ and $src(E) = type(src(e_i))$ and $tar(E) = type(tar(e_i))$. Otherwise, several adapters need to be given for each combination of subclasses in $clan(src(E))$ and $clan(tar(E))$. Condition 3 ensures that, given each attribute type A, all possible values for such attribute are considered by the binding, for each subclass of the owner of A. As in the previous case, such adapters can be given for $src(A)$ or for its subclasses in $clan(src(A))$. With respect to covering of values, if we have a boolean attribute, we could have either one adapter covering its two possible values, or two adapters covering one possible value each. Even for attributes admiting infinite possible values (e.g, natural numbers), this presents no problem, as adapters can be equipped with formulae over variables. Therefore for a natural number, we can use several adapters, each covering certain ranges. For example one covering $X < 0$, another covering $X = 0$, and finally one covering $X > 0$.

Adapters are amalgamated either by covering a model conformant to the concept and next merging the store components and then the meta-model instances of all adapters in the covering, or by covering a model conformant to the meta-model, and then merging the store components and the concept instances of all adapters in the covering. The first amalgamation kind (called forward amalgamation) requires forward-compatibility of adapters and will be used to transform rules defined over concepts so that they become applicable to the bound meta-models. The second amalgamation kind (called backwards amalgamation) requires backwards-compatibility and will be used to retype a meta-model instance according to a concept. Figure 21 shows a schema of both adpater usages.



$B_{C,MM}$
C
MM
type
type'
rule
rule'
M
Transformation through (forward) amalgamation
(a)
$B_{C,MM}$
C
MM
type
type'
rule
E
M
Transformation through (backwards) amalgamation
(b)

Figure 21: Forward amalgamation: using adapters to translate concept rules into meta-model rules (a). Backwards amalgamation: using adapters to translate meta-model instances into concept instances (b).

We first start defining what a covering of a binding with respect to a model is. We assume such model is conformant to the concept. Typically, such model will be the left or right hand side of a rule defined over a concept.

Definition 18 (Covering). *Given a binding $B_{C,MM} = \langle m, P\rangle$ and a model E typed on C, a* covering *is the set $COV_E = \{n_i \colon CI_i \to E \mid CI_i \xrightarrow{c_i} S_i \xleftarrow{m_i} M_i \in P\}$ of all M-morphisms from some CI_i to E. A covering is* complete *if the morphisms in COV_E are jointly surjective.*

Remark. A set of morphisms $\{n_i \colon CI_i \to E\}$ with same codomain E is jointly surjective, if every element of E is mapped by some element of some CI_i.

It is interesting to note that a complete binding with respect to C yields a complete covering over any grounded model typed by C, as next proposition formalizes.

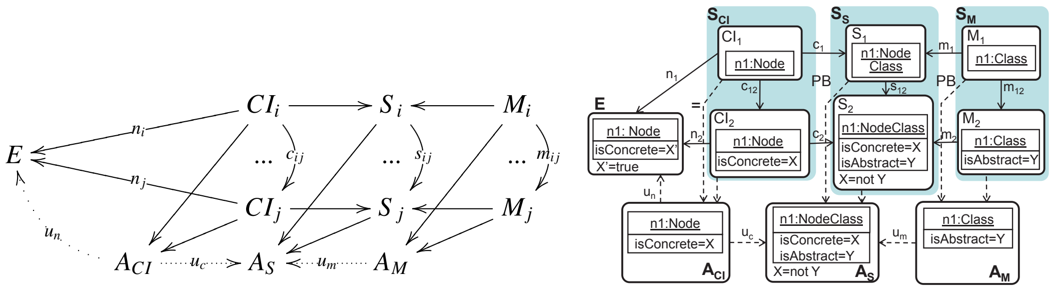


Figure 22: Adapter amalgamation (left). Amalgamation example (right).

Proposition 1 (Binding and Covering Completeness)**.** *Given a forward-compatible binding $B_{C,MM}$ that is complete with respect to C and any grounded model E typed by C, then the covering COV_E is complete.*

Proof. (Sketch) $B_{C,MM}$ is complete w.r.t. C and hence it has an adapter for each node type and edge type in C. Therefore the covering is complete w.r.t. the nodes and edges in E. Moreover, there are adapters covering every possible value for each attribute type in C, and hence the covering is also complete w.r.t. attributes in E. Please note that if M is not grounded, it is not possible to ensure complete coverings even for complete bindings. $\square$

Completeness of a binding with respect to a concept C yields a complete covering for any grounded model typed by C. However, it may not yield a complete covering for grounded models typed by MM (for which we would need completeness w.r.t. MM). Hence, given a binding $B_{C,MM}$, which is complete with respect to a concept C, and a model M typed by MM, the covering COV_M with respect to M may not be complete. This is so as in general, MM may contain elements not considered by the binding (intuitively, MM might be "bigger" than the concept C), and hence if a model M contains instances of those elements, COV_M will not be complete.

Next, we present amalgamation, a mechanism to glue together adapters, to build a bigger one. The idea is to cover a model E, conformant to a concept C, or a model M, typed by MM. The next definition corresponds to forward-amalgamation, covering a concept instance. Backward amalgamation (covering meta-model instances) is symmetrical, but requires backwards-compatibility of the binding.

Definition 19 (Forward Amalgamation)**.** *Given a covering COV_E and a forward-compatible binding $B_{C,MM}$, an amalgamation is a co-span $A_{CI} \xrightarrow{u_c} A_S \xleftarrow{u_m} A_M$, constructed as follows (see the left of Figure 22):*

- *Let $D_{CI} = \langle Ob_{CI}, Mor_{CI} \rangle$ be a diagram made of a* bag *of objects $Ob_{CI} = \{CI_i \mid \exists n_i \colon CI_i \rightarrow E \in COV_E\}$ and a set of morphisms $Mor_{CI} = \{c_{ij} \colon CI_i \rightarrow CI_j \mid CI_i \xrightarrow{c_i} S_i \xleftarrow{m_i} M_i \in P \wedge CI_j \xrightarrow{c_j} S_j \xleftarrow{m_j} M_j \in P \wedge\ c_{ij}$ injective $\wedge\ \exists n_i, n_j \in COV_E$ s.t. $n_i = n_j \circ c_{ij}\}$.*
- *A_{CI} is the co-limit of the diagram D_{CI}, and $u_n \colon A_{CI} \rightarrow E$ exists due to the co-limit universal property.*
- *Let $D_S = \langle Ob_S, Mor_S \rangle$ be a diagram made of a* bag *of objects $Ob_S = \{S_i \mid CI_i \xrightarrow{c_i} S_i \xleftarrow{m_i} M_i \in P \wedge n_i \colon CI_i \rightarrow E \in COV_E\}$, and a set of morphisms $Mor_S = \{s_{ij} \colon S_i \rightarrow S_j \mid CI_i \xrightarrow{c_i} S_i \xleftarrow{m_i} M_i \in P \wedge CI_j \xrightarrow{c_j} S_j \xleftarrow{m_j} M_j \in P \wedge s_{ij}$ injective $\wedge\ \exists c_{ij} \in D_{CI}$ s.t. $s_{ij} \circ c_i = c_j \circ c_{ij}$ and the square is PB$\}$.*

- *A_S is the co-limit object of D_S, by the co-limit universal property $\exists_1 A_{CI} \overset{u_c}{\to} A_S$.*
- *Let $D_M = \langle Ob_M, Mor_M \rangle$ be a diagram made of a* bag *of objects $Ob_M = \{M_i \mid CI_i \overset{c_i}{\to} S_i \overset{m_i}{\leftarrow} M_i \in P \wedge n_i \colon CI_i \to E \in COV_E\}$, $Mor_M = \{m_{ij} \colon M_i \to M_j \mid CI_i \overset{c_i}{\to} S_i \overset{m_i}{\leftarrow} M_i \in P \wedge CI_j \overset{c_j}{\to} S_j \overset{m_j}{\leftarrow} M_j \in P \wedge m_{ij}$ injective $\wedge\ \exists s_{ij} \in D_S$ s.t. $s_{ij} \circ m_i = m_j \circ m_{ij}$ and the square is PB$\}$.*
- *A_M is the co-limit object of D_M, by the co-limit universal property $\exists_1 A_M \overset{u_m}{\to} A_S$.*

Remark. The diagrams D_{CI}, D_S and D_M contain bags of objects, because there can be several occurrences of the same object (i.e., repetitions are allowed). The reason is that the same adapter can be used several times in the covering COV_E. In addition, please note that the set of morphisms of a diagram can be empty.

Remark. The previous procedure can also be used to build an amalgamation for a meta-model instance M (backwards amalgamation). For this purpose, a covering COV_M and a backwards-compatible binding $B_{C,MM}$ are needed.

Example 18. The right of Figure 22 shows a simple amalgamation, where the left parts of each adapter cover the model E. The squares are PBs due to adapters compatibility.

The amalgamation of several forward-compatible (resp. backwards-compatible) adapters yields an adapter that is forward-compatible (resp. backwards-compatible) with them. We will use this result in Section 5 when instantiating rules defined over concepts for specific meta-models, using adapters.

Theorem 1 (Compatibility of amalgamation)**.** *Given a covering COV_E and a forward-compatible binding $B_{C,MM}$, the resulting amalgamation $A_{CI} \overset{u_c}{\to} A_S \overset{u_m}{\leftarrow} A_M$ is forward-compatible with the adapters in $B_{C,MM}$.*

Proof. In appendix. □

5. Using Algebraic Adapters

Given a transformation defined over a concept and a binding from the concept to a certain meta-model MM, we can use the binding adapters both for *meta-model adaptation* (i.e. extending the meta-model instances with the necessary information to be able to apply the transformation on them) and *template instantiation* (i.e. adapting the transformation to enable its direct application on the meta-model instances), as shown in Figure 21. The next subsections illustrate these two approaches.

5.1. Transforming models: Meta-model adaptation approach

Given a model M conforming to a meta-model bound to the concept, we can use the adapters in the binding to build the store and concept models S and E associated to M, trough backwards amalgamation. This way, once retyped and with appropriate synchronization mechanisms to update the derived information, the original rules can be applied to S. The next subsection will show how to obtain rules working on S and emulating this mechanism; in this section, we will focus on the model adaptation procedure.

The diagram to the left of Figure 23 shows how to use an adapter as a co-span GT rule [13] to adapt a model M into a model E[2]. Thus, given a backwards-compatible binding $B_{C,MM}$, we

[2]Please note that we perform the amalgamation on the other side, compared to Figure 22.

$$\begin{tikzcd}
A_{CI} \arrow[r, "u_c"] \arrow[d, "m_{ci}"'] \arrow[dr, phantom, "P.O."] & A_S \arrow[d, "m_s"] \arrow[dr, phantom, "P.O."] & A_M \arrow[l, "u_m"'] \arrow[d, "m_m"] \\
E \arrow[r] & S & M \arrow[l]
\end{tikzcd}$$

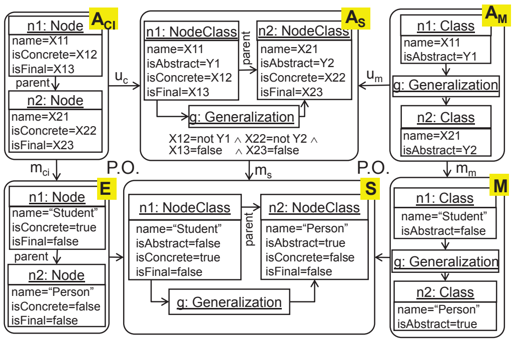


Figure 23: Model adaptation schema (left). Model adaptation example (right).

build a covering COV_M for the model M and an amalgamation $A_{CI} \to A_S \leftarrow A_M$, together with the morphism $m_m \colon A_M \to M$ (see Figure 23). Then, the right PO creates a store model S which adds to M all elements present in the concept but not in the model, and the left PO removes from the store the elements that do not belong to the concept, yielding a model E.

Example 19. The right of Figure 23 shows the adaptation of a model M induced by a binding. The amalgamation $A_{CI} \to A_S \leftarrow A_M$ results from the covering of model M by the adapters in the binding (in particular, there are M-morphisms from the meta-model component of the adapters in Figures 17(b–e) to M). The store model in the middle contains the elements in M plus all the extra information. It can be seen as a virtual view gathering the information that is not in the model, but is needed by the concept. In the lower models, we have used a shortcut for attribute values, e.g., we write *isFinal=false* instead of *isFinal*=X and the formula X=*false*.

Models M, S and E are typed with respect to different meta-models (MM, MM_S and C respectively). However, if we adapt M following the scheme in Figure 23, the resulting model E may still include elements not typed by C in certain cases, in particular, if the binding is not complete with respect to MM (i.e. if there are elements in MM which do not participate in the binding). These extra elements need to be discarded in advance, as shown in Figure 24. Hence, in a first step, we consider only the submodel $M' \hookrightarrow M$ that is completely covered by COV_M. The following Definition 20 describes the model adaptation and how the typing is calculated.

Definition 20 (Model adaptation)**.** *Let C be a concept and $B_{C,MM} = \langle m, P\rangle$ a backwards-compatible binding. Given a model M typed over MM by $type_M \colon M \to MM$, and a covering COV_M over M, M is adapted to the typing of concept C by the diagram in Figure 24, where the amalgamation $A_{CI} \overset{u_c}{\to} A_S \overset{u_m}{\leftarrow} A_M$ is built according to Definition 18, and M' is the submodel of M which is completely covered by COV_M.*

The typing $type_S$ uniquely exists due to the pushout universal property, while $type_E$ is simply $type_{A_{CI}} \circ m_{ci}^{-1}$.

Remark. The left pushout in Figure 24 is calculated as a pushout complement. As m_m is surjective, so is m_s (and m_{ci}) [4], and therefore no dangling edges can occur. However, the identification condition may occur, and the pushout fail to exist, if m_m is not injective. Therefore, from now one we assume injective matches for the adapters.

Example 20. Figure 25 shows an example of model adaptation using a covering that is not complete. Let us assume that the UML2 meta-model in Figure 16 includes the type *Operation*, but this type is not considered by any adapter in the binding (basically, because our concept does

$$\begin{tikzcd}
A_{CI} \arrow[r, "u_c"] \arrow[d, "m_{ci}"'] \arrow[dr, phantom, "P.O."] & A_S \arrow[d, "m_s"] \arrow[dr, phantom, "P.O."] & A_M \arrow[l, "u_m"'] \arrow[d, "m_m"] \arrow[ddd, bend left, "type_{A_M}"] \\
E \arrow[r] \arrow[dd, "type_E"] & S \arrow[dd, dotted, "type_S"] & M' \arrow[l] \arrow[d, hook] \\
 & & M \arrow[d, "type_M"] \\
C \arrow[r] & MM_S & MM \arrow[l]
\end{tikzcd}$$

Figure 24: Typing an adapted model.

not include such a notion). Hence, a model M can include an *Operation* object because it is allowed by the meta-model, like model M in the figure. If we do not discard such object by taking the submodel M', the pushout would copy such object to the store S and E models.

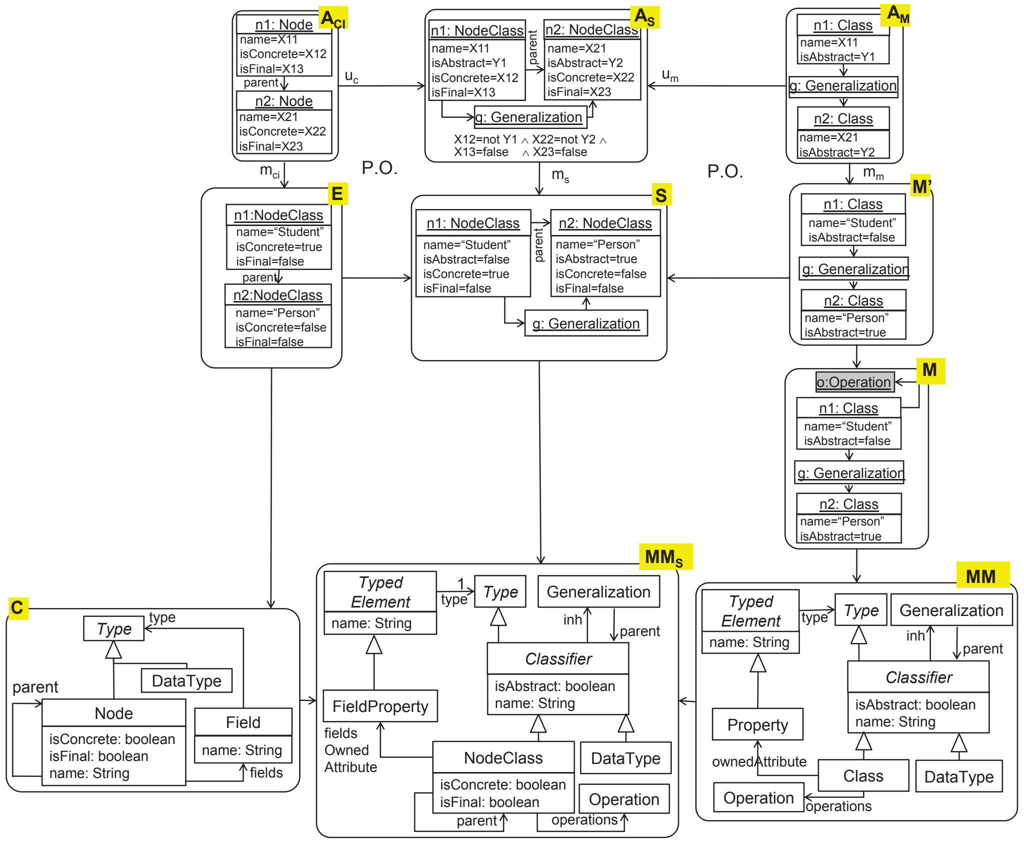


Figure 25: Model adaptation considering a non-complete covering of M.

In this example, $M \cong M'$ if the binding is complete with respect to the UML2 meta-model and includes some adapter making use of an *Operation* object.

Once the model M is adapted, if the modelling framework has suitable synchronization mechanisms for the derived information, we can retype and adapt the original rules, so that they become applicable on the model S. The next section shows how to perform this rule adaptation and obtain synchronization mechanisms.

5.2. Transforming rules: Template instantiation approach

In the previous subsection, transformations defined over concepts become applicable to the instances of the bound meta-models by adapting such instances. In this subsection, we present an alternative approach to transformation reuse, where we adapt the transformation rules defined over the concept so that they can be directly applied on instances of the bound meta-models.

We consider Double Pushout GT rules [11]. These are made of a span $L \xleftarrow{l} K \xrightarrow{r} R$ of M-morphisms with a left-hand side model L, a right hand-side model R, and a kernel model K accounting for the preserved elements. Models L, K and R do not need to be grounded. L contains the elements that need to be found in a host model M to enable the rule application, $L \setminus K$ contains those that should be deleted, while $R \setminus K$ contains the ones to be created. The match $m\colon L \to M$ is an M-morphism as well. For the sake of simplicity, we do not consider rule application conditions.

We call rule templates to rules typed by a concept.

Definition 21 (Rule Template)**.** *Given a concept C, a rule template $L \xleftarrow{l} K \xrightarrow{r} R$ is a Double Pushout rule with injective l and r, typed over C.*

Next, we show how to adapt a rule template $L_C \xleftarrow{l_C} K_C \xrightarrow{r_C} R_C$ defined over a concept, into a rule $L_M \xleftarrow{l_M} K_M \xrightarrow{r_M} R_M$ defined over a meta-model bound to the concept. In the process, we also obtain a rule $L_S \xleftarrow{l_S} K_S \xrightarrow{r_S} R_S$ working on the store model, which can be seen as a mechanism to synchronize the "real" model elements and the derived ones, as well as an integrated rule $\langle L_C \to L_S \leftarrow L_M\rangle \xleftarrow{\langle l_C, l_S, l_M\rangle} \langle K_C \to K_S \leftarrow K_M\rangle \xrightarrow{\langle r_C, r_S, r_M\rangle} \langle R_C \to R_S \leftarrow R_M\rangle$ to modify the three models in the co-span at the same time.

Rule adaptation procedure

Inputs:

- A forward-compatible binding $B_{C,MM}$ complete w.r.t. C
- rule template $L_C \xleftarrow{l_C} K_C \xrightarrow{r_C} R_C$ typed over the concept C.

Outputs:

- adapted rule $L_M \xleftarrow{l_M} K_M \xrightarrow{r_M} R_M$ typed on MM.
- adapted rule $L_S \xleftarrow{l_S} K_S \xrightarrow{r_S} R_S$ typed on the store.
- integrated rule $\langle L_C \to L_S \leftarrow L_M\rangle \xleftarrow{\langle l_C, l_S, l_M\rangle} \langle K_C \to K_S \leftarrow K_M\rangle \xrightarrow{\langle r_C, r_S, r_M\rangle} \langle R_C \to R_S \leftarrow R_M\rangle$

Procedure:

$$\begin{tikzcd}
& C_L \arrow[rr] \arrow[dd, "m_1"] & & S_L \arrow[dd] & & M_L \arrow[ll] \arrow[dd] \\
C_K \arrow[ur] \arrow[rr] \arrow[dd, "m_2"] & & S_K \arrow[ur] \arrow[dd, "m_k"] & & M_K \arrow[ll] \arrow[ur] \arrow[dd] & \\
& L_C \arrow[rr] & & L_S & & L_M \arrow[ll, dotted, "u_2"] \\
K_C \arrow[ur] \arrow[rr] & & K_S \arrow[ur, dotted, "u_1"] & & K_M \arrow[ll] \arrow[ur] &
\end{tikzcd}
\qquad
\begin{tikzcd}
& C_K \arrow[rr] \arrow[dl] \arrow[dd] & & S_K \arrow[dl] \arrow[dd] & & M_K \arrow[ll] \arrow[dl] \arrow[dd] \\
C_R \arrow[rr] \arrow[dd] & & S_R \arrow[dd] & & M_R \arrow[ll] \arrow[dd] & \\
& K_C \arrow[rr] \arrow[dl] & & K_S \arrow[dl, dotted, "u_1"] & & K_M \arrow[ll] \arrow[dl] \\
R_C \arrow[rr] & & R_S & & R_M \arrow[ll, dotted, "u_2"] &
\end{tikzcd}$$

Figure 26: Rule template instantiation: Adapting L_C, K_C (left) and R_C (right).

1. First, we flatten the rule template (see Definition 15 in [5]) according to the chosen concept closure.
2. Next, we adapt the flattened rule template, as sketched in the left diagram of Figure 26:
   - 2.1. First, we adapt the components L_C and K_C of the rule template as follows:
     - 2.1.1. We find three complete compatible amalgamations for L_C, K_C and R_C, which we denote $C_x \rightarrow S_x \leftarrow M_x$ for $x = \{L, K, R\}$. Initially, the upper-most squares in the diagram are PBs because $C_L \rightarrow S_L \leftarrow M_L$ and $C_K \rightarrow S_K \leftarrow M_K$ are compatible. The square C_L, C_K, K_C, L_C is a PB, because the amalgamation is complete.
     - 2.1.2. We make the PO of the left-back square, leading to the PO object L_S.
     - 2.1.3. We make the PO of the left-front square, leading to the PO object K_S.

     By the PO universal property, $\exists_1 u_1 \colon K_S \rightarrow L_S$. The left cube is a van Kampen square with left and top squares PBs, back and front faces POs; therefore, the squares S_K, S_L, L_S, K_S and K_C, L_C, L_S, K_S are PBs.
   - 2.2. Then, we build the right cube of the same diagram as follows:
     - 2.2.1. We construct the right-front square by calculating the PO complement K_M, so that the square S_K, M_K, K_S, K_M becomes a PO. Such PO complement exists because, being the amalgamations done through complete coverings, m_k is surjective, and hence no dangling edges can occur.
     - 2.2.2. We make the PO of the right-most square, leading to the PO object L_M.

     By the PO universal property, there is a unique $u_2 \colon L_M \rightarrow L_S$. The square S_L, M_L, L_M, L_S is a PO because the right cube is a van Kampen square, and we have that the front face is a PO, the left and top faces are PBs, and hence the back face is a PO.

     Note that K_M exists if C_K is a complete covering of K_C. This is so as, in such a case, $m_2 \colon C_K \rightarrow K_C$ is an epimorphism, and so is $m_k \colon S_K \rightarrow K_S$. Therefore, the dangling edges condition is not applicable [11], and the PO complement K_M should exist.
3. Finally, we adapt the R_C component in the rule template and build its relation with the previously defined K components, as sketched in the right diagram of Figure 26. As before:
   - 3.1. We construct the left cube (the back face is a PO as explained before). For this purpose, we first build the front face as the PO of $R_C \leftarrow C_R \rightarrow S_R$, yielding R_S. By the PO universal property of the back PO, $\exists_1 u_1 \colon K_S \rightarrow R_S$. The left cube is a van Kampen square, with back and front faces POs, left and top faces PBs, and so the squares S_K, S_R, K_S, R_S and K_C, K_S, R_C, R_S are PBs.

3.2. We construct the right-most square as the PO of $M_R \leftarrow M_K \rightarrow K_M$, yielding R_M. By the PO universal property, $\exists_1 u_2 \colon R_M \rightarrow R_S$. The right cube is a van Kampen square, where the back face is a PO, the top and left faces are PBs, and the front face is a PO, therefore the bottom and right faces are PBs.

Example 21. Figure 27 shows the integrated rule $\langle L_C \rightarrow L_S \leftarrow L_M\rangle \xleftarrow{\langle l_C,l_S,l_M\rangle} \langle K_C \rightarrow K_S \leftarrow K_M\rangle \xrightarrow{\langle r_C,r_S,r_M\rangle} \langle R_C \rightarrow R_S \leftarrow R_M\rangle$ that results from adapting the rule template $L_C \leftarrow K_C \rightarrow R_C$. The integrated rule synchronizes the original rule (top row), a rule working on the store model (middle row) and a rule working on instances of the bound meta-model (lower row). According to the construction, all squares are pullbacks.

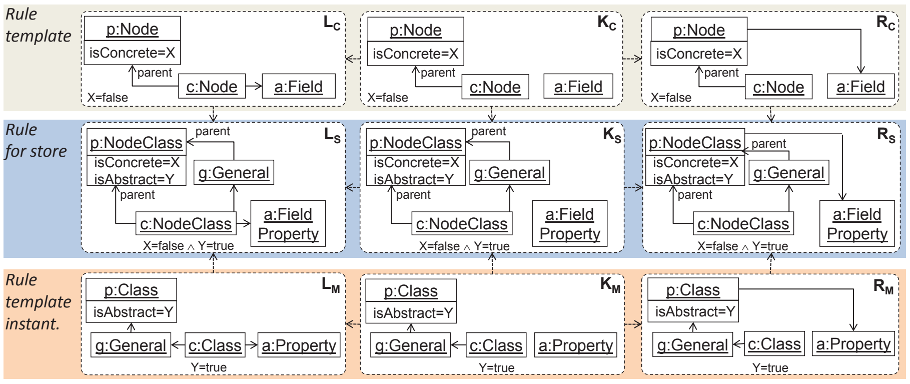


Figure 27: Example of rule template instantiation: *Pull-up field* refactoring.

Please note that having a complete binding over the concept does not guarantee a complete covering of a rule's LHS. This is so because the LHS is not a grounded model. It may contain arbitrary expressions in its formula, which may prevent from finding an M-morphism from the adapter to the LHS. In some cases, this problem can be solved by generating, from the "problematic" rule, a set of more concrete rules (with grounded models in LHS) for which we can obtain complete covering and which altogether consider all possible mappings modelled by the binding adapters. We leave an in-depth study of this concretization technique for future work, and here we just sketch it with an example

Example 22. Figure 28 shows an example of rule template concretization. Compartment (a) shows two adapters which cover the two possible values for attribute *isConcrete*. Figure 28(b) shows a rule template that we want to instantiate. As its LHS is not grounded (it contains the expression $X \neq Y$), there are no M-morphisms from the left part of the adapters to the LHS, and hence there is no complete covering of the LHS by the binding. In this situation, we can concretize the rule with respect to the ground values of the adapters as follows. First, we concretize the variable X in the LHS, by creating one rule for each possible value of X. This yields two possible rules with formulas $X = true \wedge X \neq Y$ and $X = false \wedge X \neq Y$. Then, we concretize these two rules for variable Y, obtaining four rules (i.e. all combinations of $true$ and $false$ for X and Y). Finally, all rules with unsatisfiable formulas can be neglected, obtaining the two concretized rules shown in Figure 28(c).

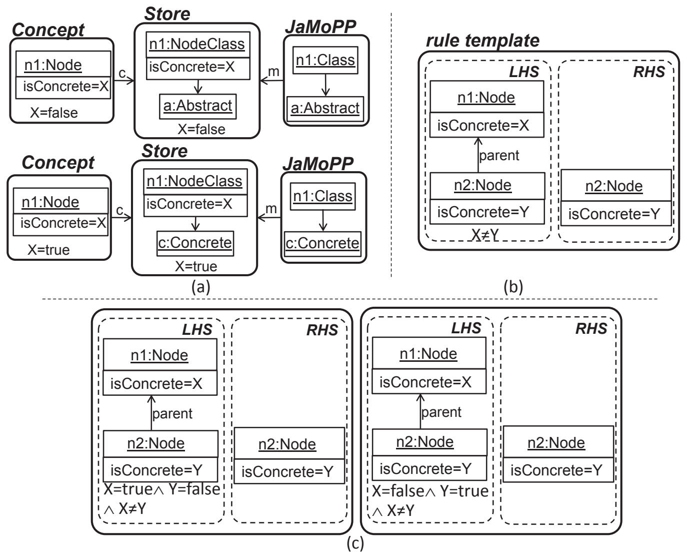


Figure 28: Example of rule concretization. (a) Adapters for attribute *isConcrete*. (b) Rule template. (c) Two template rule concretizations.

5.3. Behaviour preservation: Correctness and completeness

The instantiation of a transformation template for a binding needs to preserve the original behaviour of the template. Figure 29 depicts the needed correctness conditions. Starting from a model M', we calculate its store and concept models $E \to S \leftarrow M$, taking the maximal submodel M of M' for which there is a complete covering. The behaviour of the instantiated GT template is complete if it leads to all behaviours of the original template. That is, if for each possible transformation of E using the original rules ($E \Rightarrow^{trf^*} E''$), there is a transformation $trf_m{}^*$ using the adapted rules starting from M and yielding M'', a corresponding transformation $M' \Rightarrow^{trf_m{}^*} M'''$, and an integrated transformation $trf_i{}^*$ that transforms the co-spans yielding the co-span $E'' \to S'' \leftarrow M''$. Correctness means that the instantiation does not produce extra behaviour: each transformation $trf_m{}^*$ has a corresponding transformation trf^*.

To demonstrate correctness and completeness, we proceed in three steps. First, we check equivalence of rule matches; then, we check rule applicability; and finally, we argue about rule sequences.

The next proposition states the equivalence of matches of the three rules making an integrated rule, provided that the host model M for the matches was adapted using the amalgamation procedure presented in subsection 5.1. This equivalence shows the compatibility of rule and model amalgamation. Similar to matches for amalgamation, in the rest of the paper, we assume injective matches for rules.

Lemma 1 (Match equivalence). *Given an integrated rule $\langle L_C \to L_S \leftarrow L_M \rangle \xleftarrow{\langle l_C, l_S, l_M \rangle} \langle K_C \to K_S \leftarrow K_M \rangle \xrightarrow{\langle r_C, r_S, r_M \rangle} \langle R_C \to R_S \leftarrow R_M \rangle$ constructed as explained before, and a cospan $E \to S \leftarrow M$ resulting from adapting M, the following statements are equivalent:*
(i) $\exists m_m\colon L_M \to M$, (ii) $\exists m_s\colon L_S \to S$, (iii) $\exists m_c\colon L_C \to E$.

$$\begin{tikzcd}
C \arrow[r] & MM_S & MM \arrow[l] & \\
A_{CI} \arrow[u, "type_{A_{CI}}"] \arrow[r, "u_c"] \arrow[d, "m_{ci}"'] \arrow[dr, phantom, "P.O."] & A_S \arrow[u, "type_{A_S}"] \arrow[d, "m_s"'] \arrow[dr, phantom, "P.O."] & A_M \arrow[l, "u_m"'] \arrow[u, "type_{A_M}"] \arrow[d, "m_m"'] & \\
E \arrow[r] \arrow[d, Rightarrow, "trf*"'] & S \arrow[uu, dotted, bend right, "type_S"'] \arrow[d, Rightarrow, "trf_i^*"'] & M \arrow[l] \arrow[r, hook] \arrow[uu, bend right, "type_M"'] \arrow[d, Rightarrow, "trf_m^*"'] & M' \arrow[uul, bend right, "type_{M'}"'] \arrow[d, Rightarrow, "trf_{m'}^*"'] \\
E'' \arrow[r] & S'' & M'' \arrow[l] \arrow[r, hook] & M'''
\end{tikzcd}$$

Figure 29: Correctness and completeness conditions.

Proof. In appendix. □

Example 23. The upper two rows in Figure 30 show an example of match equivalence. By construction, the existence of $m_m : L_M \to M$ implies the existence of $m_s : L_S \to S$ and $m_c : L_C \to E'$. In this case, $M \cong M'$.

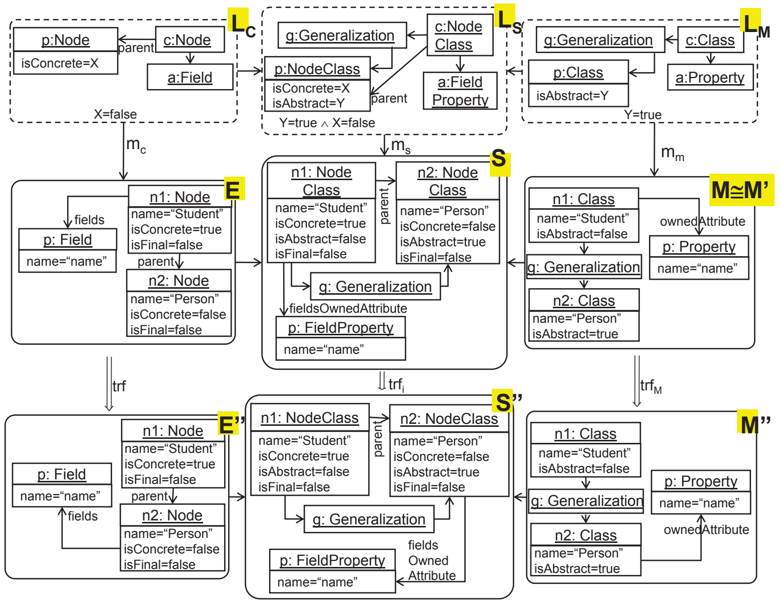


Figure 30: Match and applicability equivalence example (upper and middle row). And application equivalence example (bottom row).

Example 24. Figure 31(a) shows two adapters, the second one mapping edge *parent* in the concept to nothing in the model. As Figure 31(b) illustrates, this adapter has the effect of removing the edge from any adapted rule that is instantiated from a rule template (i.e. the adapter is used in a forward amalgamation, left-to-right), or adding an edge between any pair of classes when a model M is adapted into a model E (i.e. the adapter is used in a backwards amalgamation, right-to-left). The co-span $E \rightarrow S \leftarrow M$ shown in Figure 31(b) is calculated as in Definition 20 (i.e., E is built from M by backwards amalgamation), and there are 6 M-morphisms from the LHS of the template rule into E, and 6 from the LHS of the adapted rule into M (i.e. the matches are preserved). Still, we will see later that this kind of adapters may introduce unforeseen dangling edges. Figure 31(c) shows a different co-span that results from building M from E (i.e. the other way round). In this case, matches are not preserved, as there is 1 match from the LHS of the template rule into E, but 6 matches from the LHS of the adapted rule into M. However, this co-span is not constructed as explained in Definition 20 (starting from M), which is required by our notion of match equivalence.

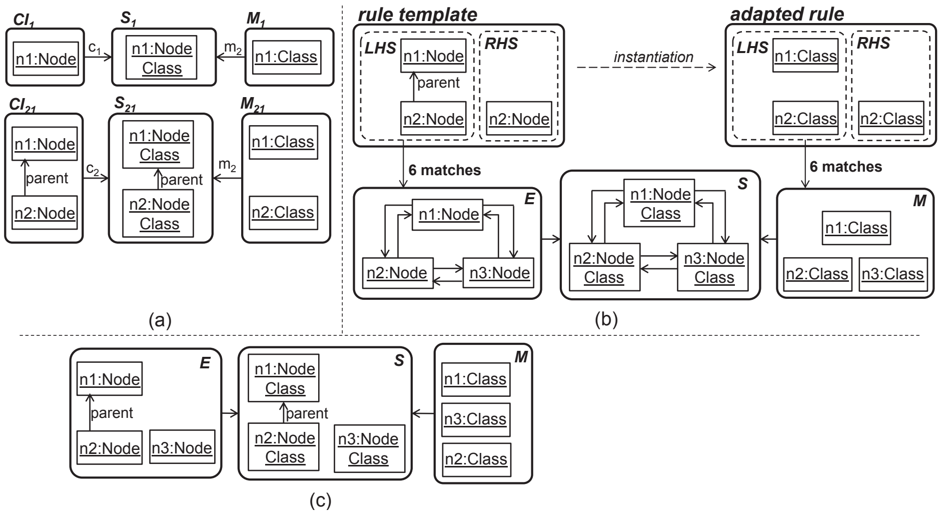


Figure 31: (a) Two adapters. (b) The adapters are used to instantiate a rule template and to build model E from model M. (c) Building model M from model E may not fulfill match equivalence.

Given that a certain rule template is applicable in E, the instantiated rule may fail to be applicable in M due to the dangling edge condition. In particular, this may happen if the meta-model MM defines an edge type r that the binding does not map, the model M defines instances of the edge, and the rule deletes the source or target objects of the edge. This corresponds to the situations depicted in Figure 32(b): some node in the concept is bound, but the mapped meta-model node has incident edges which are not considered by the binding. In this case, if some rule deletes nodes of type N, we should check whether the particular input model M contains N instances with adjacent edges of type r, otherwise we may not have the same rule applicability. Additionally, these new edges in MM should be checked for ancestors or descendants of N. Checking whether a particular model M has instances of the problematic edges can be implemented using graph constraints [11] like those in Figure 32(c).

Definition 22 (Deletion-safe binding and model)**.** *Given a binding* $B_{C,MM} = \langle m\colon \overline{C'} \rightarrow MM, P\rangle$*,*

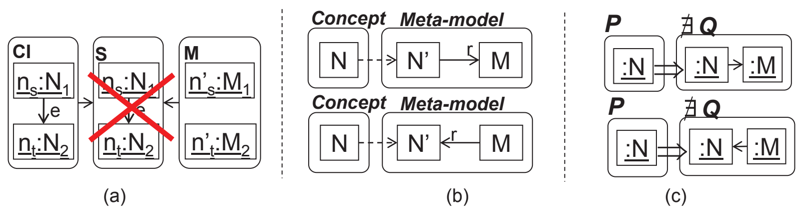


Figure 32: (a) Invalid adapter. (b) Completeness condition for bindings is ensured if no such bindings exist (i.e. node N mapped and incident edge unmapped) when some rule deletes nodes of type N. (c) Completeness condition for initial models, when the template rule deletes nodes of type N.

and a co-span $C \xrightarrow{c} S \xleftarrow{m} MM$, we say that $B_{C,MM}$ is T-deletion-safe for $T \in V_C$, if:

$$\forall e \in E_{MM}[src_E(e) \in clan(c(T)) \vee c(T) \in clan(src_E(e)) \vee \\ tar_E(e) \in clan(c(T)) \vee c(T) \in clan(tar_E(e))] \Rightarrow \exists e' \in E_C \text{ with } c(e') = m(e)$$

We call $unsafe(T)$ to the set of all edges e not satisfying the previous condition for a node T. Given a binding $B_{C,MM} = \langle m\colon \overline{C'} \to MM, P\rangle$ and a model M with $type\colon M \to MM$, we say that M is safe if $\forall T \in V_C \nexists e \in E_M$ with $type(e) \in unsafe(T)$.

Similarly, adapters mapping an edge $n_s \xrightarrow{e} n_t$ to a structure where n_s and n_t are mapped to unconnected n'_s and n'_t (like in Figures 31(a) and 32(a)) are problematic. This is so as a template rule deleting a node of type $type(n_s)$ might fail in E due to the dangling edge condition, but would be applicable in M as n'_s lacks adjacent edges. Hence, edges should be mapped similar to Figure 17(c). The converse situation, where an edge in M is not mapped to an edge in the concept is also forbidden, as it might make a rule applicable in E but no applicable in M due to dangling edges.

Definition 23 (Invalid adapter)**.** *An adapter $CI_i \xrightarrow{c_i} S_i \xleftarrow{m_i} M_i$ is invalid if:*

- *$\exists e \in E_{CI_i}$ s.t. $\nexists e' \in E_{M_i}$ with $m_i(e') = c_i(e)$ and $\texttt{disconnected}(s', t')$ with $m_i(s') = c_i(src(e))$ and $m_i(t') = c_i(tar(e))$, or*
- *$\exists e \in E_{M_i}$ s.t. $\nexists e' \in E_{CI_i}$ with $c_i(e') = m_i(e)$ and $\texttt{disconnected}(s', t')$ with $c_i(s') = m_i(src(e))$ and $c_i(t') = m_i(tar(e))$.*

The `disconnected` *predicate holds if the two parameter nodes do not belong to the same connected component.*

Now we are ready to state the conditions for applicability equivalence of a template rule and the derived rule, as expressed by Lemma 2.

Lemma 2 (Applicability equivalence)**.** *Given an integrated rule $\langle L_C \to L_S \leftarrow L_M\rangle \xleftarrow{\langle l_C, l_S, l_M\rangle} \langle K_C \to K_S \leftarrow K_M\rangle \xrightarrow{\langle r_C, r_S, r_M\rangle} \langle R_C \to R_S \leftarrow R_M\rangle$ derived from a binding without invalid adapters, a safe model M' and a match $m_m\colon L_M \to M$, the following statements are equivalent:*

(i) rule $L_M \leftarrow K_M \to R_M$ is applicable to M' at m_m,

(ii) *rule* $L_M \leftarrow K_M \rightarrow R_M$ *is applicable to* $M \hookrightarrow M'$ *at* m_m,

(iii) *rule* $L_S \leftarrow K_S \rightarrow R_S$ *is applicable to* S *at* m_s *(the equivalent match of* m_m *for model* S*)*

(iv) *rule* $L_C \leftarrow K_C \rightarrow R_C$ *is applicable to* E *at* m_c *(the equivalent match of* m_m *for model* E*)*

Proof. In appendix. □

Remark. The condition for invalidity of adapters of Definition 23 are sufficient but not necessary. One might employ here the conditions for the classical embedding theorem for graph transformation [11], but those conditions need to be checked "a posteriori", after a particular transformation has been applied. In our case, we can check the conditions independent on the particular transformation to be applied.

Example 25. Figure 31(a) showed an example of invalid adapter (the bottom adapter is invalid, as it does not satisfy Definition 23). In Figure 31(b), there are six matches from the LHS of the template rule to E and from the LHS of the adapted rule to M (match equivalence). However, the template rule cannot be applied in any of the matches due to the dangling edge condition, whereas the adapted rule can be applied in all matches (i.e. there is no applicability equivalence).

On the contrary, we have applicability equivalence if we use valid adapters, as illustrated in Figure 33. Compartment (a) shows the adapters used, while (b) shows the template and adapted rules, both with equivalent applicability. Both rules are applicable at equivalent matches in the co-span shown in Figure 33(c), as the starting model M is safe. In unsafe models, like M' in the co-span (d), the original rule may be applicable, but the adapted rule might be not due to the dangling edge condition. In this example, the edge type connecting *Class* and *Operation* is unsafe because it does not participate in the binding. Therefore, model M' is unsafe because it contains an instance of such an edge.

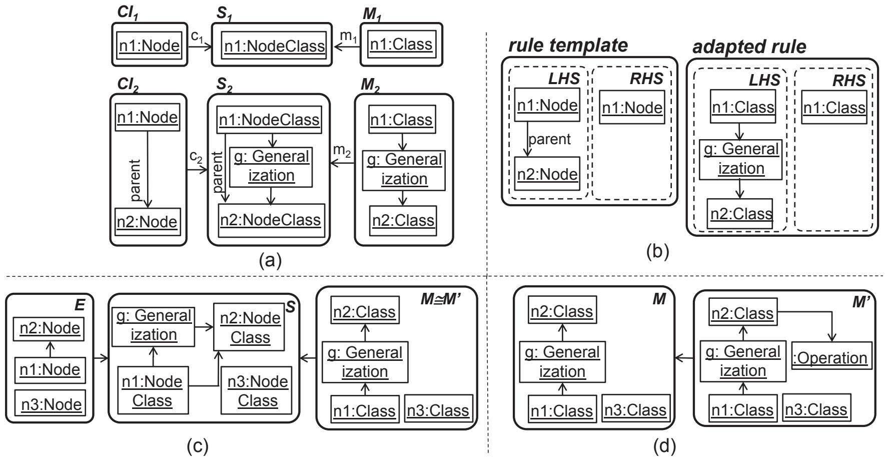


Figure 33: (a) Adapters. (b) The adapters are used to instantiate a rule template. (c) Applicability equivalence for the template and adapted rules, in a safe model M. (d) Unsafe model M.

Finally, the last step is to show that applying the original and derived rules on equivalent matches leads to equivalent results. Lemma 3 states the equivalence of the result of applying rules derived from bindings with no invalid adapters, to safe models M.

$$\begin{tikzcd}
 & & L_M \arrow[dl] \arrow[ddd, "m_m"] & & K_M \arrow[ll, "l_M"'] \arrow[rr, "r_M"] \arrow[dl] \arrow[ddd] & & R_M \arrow[dl] \arrow[ddd] \\
 & L_S \arrow[ddd, "m_s"] & & K_S \arrow[ll, "l_S"'] \arrow[rr, "r_S"] \arrow[ddd] & & R_S \arrow[ddd] & \\
L_C \arrow[ur] \arrow[ddd, "m_c"'] & & K_C \arrow[ll, "l_C"'] \arrow[rr, "r_C"] \arrow[ur] \arrow[ddd] & & R_C \arrow[ur] \arrow[ddd] & & \\
 & & M \arrow[dl] & & M_D \arrow[ll] \arrow[rr] \arrow[dl, dotted, "m_d"] & & M'' \arrow[dl, dotted, "m'"] \\
 & S & & S_D \arrow[ll] \arrow[rr] & & S'' & \\
E \arrow[ur] & & E_D \arrow[ll] \arrow[rr] \arrow[ur, dotted, "s_d"] & & E'' \arrow[ur, dotted, "s'"] & &
\end{tikzcd}$$

Figure 34: Transformation of M, S and E, yielding $M'' \to S'' \leftarrow E''$.

Lemma 3 (Application equivalence). *Given an integrated rule $\langle L_C \to L_S \leftarrow L_M\rangle \xleftarrow{\langle l_C, l_S, l_M\rangle} \langle K_C \to K_S \leftarrow K_M\rangle \xrightarrow{\langle r_C, r_S, r_M\rangle} \langle R_C \to R_S \leftarrow R_M\rangle$ derived from a binding without invalid adapters, a safe model M, a match $m_m\colon L_M \to M$, and three equivalent matches $m_m\colon L_M \to M$, $m_s\colon L_S \to S$ and $m_c\colon L_C \to S$ (see Figure 34) where:*

(i) rule $L_M \leftarrow K_M \to R_M$ is applied to M' at m_m yielding graph M''',

(ii) rule $L_M \leftarrow K_M \to R_M$ is applied to $M \hookrightarrow M'$ at m_m yielding graph $M'' \hookrightarrow M'''$,

(iii) rule $L_S \leftarrow K_S \to R_S$ is applied to S at m_s yielding graph S''

(iv) rule $L_C \leftarrow K_C \to R_C$ is applied to E at m_c yielding graph E''

then, $\exists_1 m_d\colon M_D \to S_D$, $\exists_1 s_d\colon E_D \to S_D$, $\exists_1 m'\colon M'' \to S''$, $\exists_1 s'\colon E'' \to S''$ s.t. the bottom squares in Figure 34 are pullbacks.

Proof. In appendix. □

Example 26. Figure 30 shows an example of application equivalence. We apply the original rule to model E yielding E'', the translated rule to M yielding M'' and the synchronized rule to S yielding S''. By the previous lemma, we uniquely obtain the inclusions $E'' \to S'' \leftarrow M''$.

Theorem 2 (Correctness and completeness). *Given an integrated rule $\langle L_C \to L_S \leftarrow L_M\rangle \xleftarrow{\langle l_C, l_S, l_M\rangle} \langle K_C \to K_S \leftarrow K_M\rangle \xrightarrow{\langle r_C, r_S, r_M\rangle} \langle R_C \to R_S \leftarrow R_M\rangle$ and an adapted model $E \to S \leftarrow M \hookrightarrow M'$ as in Figure 29, we have:*

- *Correctness: For each transformation $M' \Rightarrow^{trf_m^*} M'''$, exists $M \Rightarrow^{trf_m^*} M''$ and $E \Rightarrow^{trf^*} E''$ and $S \Rightarrow^{trf_i^*} S''$ leading to $E'' \to S'' \leftarrow M'' \hookrightarrow M'''$*
- *Completeness: For each transformation $E \Rightarrow^{trf^*} E''$, exists $M' \Rightarrow^{trf_m^*} M'''$ and $M \Rightarrow^{trf_m^*} M''$ and $S \Rightarrow^{trf_i^*} S''$ leading to $E'' \to S'' \leftarrow M'' \hookrightarrow M'''$*

Proof. (Sketch) Follows from lemmas 1, 2 and 3, and by the fact that, since correspondences are injective, the resulting rules cannot introduce new conflicts (critical pairs) that were not present in the original rule. This could happen if two nodes A and B in the concept are mapped to class C in the meta-model, and the GT template includes two rules for A and B with conflicting behaviour when both are applied to C nodes. □

6. Related Work

Next, we review works related to the different ingredients of our reutilization approach: rule adaptation, adapters, model co-evolution, the use of store meta-models merging several meta-models, and genericity for model transformations.

Rule adaptation. There are many works in the literature dealing with rule adaptation [12, 15, 25, 27]. For example, in [12], the authors prove the correct behaviour of a model transformation, where the GT rules for the behaviour of the target language are derived automatically using a HOT. In [25], the authors present graph grammar aspects, as aspect rules that modify rules in a base grammar. In [27] the author introduces grammar transformations, for the purpose of adaptation of a rule to the different levels of abstraction of the model. While in all these approaches rule modifications are expressed with rules, we model rule adaptations by means of adapters, and we show the equivalence of the template rules and adapted rules. Moreover, our goal is reutilization of rules.

Adapters. Our algebraic adapters are similar to the mapping operators (MOPs) [39], which are a way to specify transformations by connecting source and target meta-model elements. However, while the purpose of MOPs is to build model transformations, our goal is to resolve heterogeneities for the binding, and while MOPs are specified at the meta-model level, our adapters are defined at the model level.

One can think of our adapters as the duals of triple graph grammar (TGG) rules [32]. TGG rules are spans of morphisms typed by a meta-model triple (a span of meta-models). A correspondence meta-model is used to link elements in the source and target languages. Interestingly, one can build a store meta-model by the pushout of the meta-model triple, but in order to avoid information redundancy, one should first identify equivalent elements in the source and target meta-models, what we do with the binding and the adapters. However, TGG rules cannot be used in place of our adapters because we need to integrate heterogeneous information (e.g. mapping an edge to an intermediate class), whereas the middle model in TGG rules contains only traces. We believe our adapters can also serve as a way to specify bidirectional transformations, in the style of [23]. A detailed comparison of TGGs and adapter-based transformations is left for future work.

Co-evolution. In [40], the authors propose merging the original and revised meta-models into a single one to guarantee that models conformant to the original or revised meta-models are also instances of the merged meta-model. Then, model co-evolution is specified by GT rules which add or update elements according to the revised meta-model, and a check-out transformation eliminates the elements which are no longer in the revised meta-model. Also in the context of meta-model/model co-evolution, the work presented in [35] describes meta-model evolutions as co-span rules at the meta-model level, and provide a construction to derive co-span migration rules to transform instances of the original meta-model to the revised one.

Our work can be seen as a way to specify how to migrate rules typed over a concept (the original meta-model), so that they become applicable to another meta-model (the revised meta-model). Migration in our case would be given as a binding, while in co-evolution it is frequent the use of model comparison techniques to discover the changes made to the original meta-model. While a large amount of work has been done for model migration, few works deal with transformation migration yet. One of the exceptions is [15], where the authors deal with the problem of transformation evolution upon meta-model changes, providing automatic co-evolution of breaking and resolvable changes, and assistance to the transformation developer for breaking and unresolvable changes. While meta-model changes are detected automatically, we explicitly model them through adapters, enabling the automatic migration of rule conformant to the concept into rules conformant to another meta-model.

Model integration. In [9], the authors tackle the problem of relating heterogeneous models. They recognise the recurrent need to relate two models X and Y by first augmenting Y with derived information from X (yielding $Q(Y)$), which permits relating X and $Q(Y)$. This is similar to our way of relating models through an intermediate store. While they represent such relations using Kleisli categories, the possibility of using this framework in our context is subject for future work.

Meta-modelling and graph transformation. In [20, 21] graph transformation is used at the meta-model level. For this purpose, I-Graphs (graphs with inheritance) and I-Graph morphisms (based on clan morphisms) are introduced. In order to ensure the existence of pushouts and pullbacks, they define subtype-reflecting morphisms. We borrowed such idea, but using models (graphs with variables and formulas) instead of graphs.

Genericity. In generic programming [17], concepts can be mapped to types either automatically when the type has all methods required by the concept, or by means of a *concept map* when it does not. The *concept map* adds such methods, based on the available methods of the type. Hence, concept maps are a way to adapt the interface offered by the type to the interface required by the concept. In this sense, they are similar to our binding adapters.

In the context of MDE, the work in [18] compares different approaches to genericity for model transformations. One option is to adapt the transformation for a concrete meta-model. Another is to make the meta-model a subtype of the *concept* that was used to define the transformation. This implies enriching the meta-model with extra *derived* features, and adapting the model accordingly to allow the execution of the transformation. In this paper, we have formalized this approach and shown the correspondence with respect to a pure transformation adaptation.

We have been working on generic model-to-model transformation using languages like ATL [30]. In this setting, we developed a language to express bindings and adapters (with no formal semantics). That approach adapts a generic transformation for a given meta-model, and it also combines a form of meta-model adaptation. This is so as in ATL transformations, one can dynamically add *helpers* to metaclasses to emulate derived attributes (like a constant value for attribute *isFinal*). This is a practical way to solve the issue of applying adapters to the expressions in rules (see Example 19).

There are also some proposals for genericity in graph transformation. VIATRA2 [2] includes generic rules where types can be rule parameters. MOFLON supports generic and reflective rules by using the Java Metadata Interface [24]. MOFLON rules can receive string parameters that can be composed to form attribute or class names, and may contain nodes that match instances of any class. In both cases, no mechanisms, like concepts, bindings and adapters exist to control the correctness of the template instantiation, and to adapt the template to related meta-models.

Multi-Level Modelling. In standard MDE approaches, engineers normally work with two meta-levels at a time: a meta-model is defined, and then instantiated at the model level. However, some researchers have analysed the benefits of modelling using multiple meta-levels [1]. In fact, multi-level modelling could be used as a way to achieve reusability, by defining transformations over a meta-meta-model, which then could be reused for instance meta-models [8]. Figure 35 shows a comparison between multi-level based reusability, and the concept-based reusability we have presented in this work. Figure 35(b) shows that, with multi-level modelling the relation between a meta-model and the meta-meta-model has to follow the standard typing rules. Moreover, the meta-meta-model has to exist a-priori, before the meta-models can be built. In contrast, with concept-based reusability, the relation between a concept and a meta-model is given by a binding. In this paper we have show that this is more flexible than a simple type morphism. Moreover, using concepts, meta-models can exist a priori, independently of the concept, and several concepts can be bound to the same meta-model, while in general it would not be possible to have multiple typings for a meta-model.

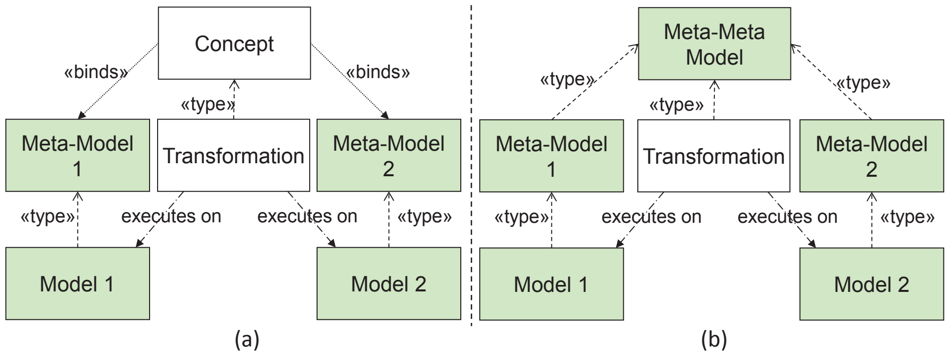


Figure 35: (a) Concept-based reusability. (b) Reusability based on multi-level modelling.

Hence, altogether, our work is novel as we consider a flexible reutilization approach, where both models and transformations can be adapted, as specified by the binding and the adapters. Our formalization enables a precise understanding of the conditions for completeness and correctness, as well as the relations between the model/meta-model adaptation [18] and the rule adaptation [6, 30] approaches to transformation reuse.

7. Conclusions and Future Work

The goal of our work is to provide a means to reuse transformations among meta-models sharing some essential features. For this purpose, we have presented an approach to specify graph transformation templates over concepts, and instantiate the templates for specific meta-models using bindings and adapters. The approach is able to resolve heterogeneities between the concept and the meta-model by the definition of an integrated model (the store). Our formalization explains the two main approaches to genericity: an adaptation of the graph transformation rules, becoming applicable to instances of the specific meta-model, and a modification of the meta-model to make it compatible with the concept (the store meta-model). We have also shown the conditions for their equivalence.

We are currently categorizing the heterogeneities we can resolve, and considering more powerful adapters, e.g. with negative application conditions. We are looking at the potential of our bindings as a simple, bi-directional transformation language, sort of dual to triple grammars [23].

We are also working on allowing non-injective correspondences, and checking the needed conditions for behaviour preservation in that case. We will also consider ways to unify concept and multi-level based reusability, as well as a more abstract formulation of the approach, based on a category of meta-models and bindings. Finally, we are also tackling richer notions of meta-models with cardinality and integrity constraints.

Acknowledgements. We thank the referees for their detailed comments, which helped to greatly improve the paper. This work has been supported by the Spanish Ministry of Economy and Competitivity with project Go-Lite (TIN2011-24139).

References

[1] C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.

[2] A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *Proc. SAC'06*, pages 1280–1287, 2006.

[3] M. Bambrilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

[4] F. Borceux. *Handbook of Categorical Algebra: Volume 1, Basic Category Theory*. Encyclopedia of Mathematics and Its Applications. Cambridge University Press, 2008.

[5] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theoretical Computer Science*, 376(3):139–163, 2007.

[6] J. de Lara and E. Guerra. Reusable graph transformation templates. In *AGTIVE'11*, volume 7233 of *LNCS*, pages 35–50, 2012.

[7] J. de Lara and E. Guerra. From types to type requirements: genericity for model-driven engineering. *Software and System Modeling*, 12(3):453–474, 2013.

[8] J. de Lara, E. Guerra, and J. Sánchez Cuadrado. Model-driven engineering with domain-specific meta-modelling languages. *Software and System Modeling*, page in press, 2013.

[9] Z. Diskin, T. S. E. Maibaum, and K. Czarnecki. Intermodeling, queries, and Kleisli categories. In *FASE*, volume 7212 of *LNCS*, pages 163–177. Springer, 2012.

[10] G. Dos Reis and J. Järvi. What is generic programming? In *Proceedings of the First International Workshop of Library-Centric Software Design (LCSD '05). An OOPSLA '05 workshop*, Oct. 2005.

[11] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer-Verlag, 2006.

[12] H. Ehrig and C. Ermel. Semantical correctness and completeness of model transformations using graph and rule transformation. In *ICGT*, volume 5214 of *LNCS*, pages 194–210, 2008.

[13] H. Ehrig, F. Hermann, and U. Prange. Cospan DPO approach: An alternative for DPO graph transformations. *Bulletin of the EATCS*, 98:139–149, 2009.

[14] Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[15] J. García, O. Díaz, and M. Azanza. Model transformation co-evolution: A semi-automatic approach. In *SLE*, volume 7745 of *LNCS*, pages 144–163. Springer, 2012.

[16] R. Garcia, J. Järvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. *SIGPLAN*, 38(11):115–134, 2003.

[17] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. *SIGPLAN Not.*, 41(10):291–310, 2006.

[18] C. Guy, B. Combemale, S. Derrien, J. Steel, and J.-M. Jézéquel. On model subtyping. In *ECMDA-FA'12*, volume 7349 of *LNCS*, pages 400–415. Springer, 2012.

[19] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. Closing the gap between modelling and Java. In *SLE'09*, volume 5969 of *LNCS*, pages 374–383. Springer, 2009.

[20] F. Hermann, H. Ehrig, and C. Ermel. Transformation of type graphs with inheritance for ensuring security in e-government networks (long version). Technical Report 2008-7, Technical University of Berlin, 2008.

[21] F. Hermann, H. Ehrig, and C. Ermel. Transformation of type graphs with inheritance for ensuring security in e-government networks. In *FASE'09*, volume 5503 of *LNCS*, pages 325–339. Springer, 2009.

[22] F. Jouault and J. Bézivin. KM3: A DSL for metamodel specification. In *FMOODS'06*, volume 4037 of *LNCS*, pages 171–185. Springer, 2006.

[23] Y. Lamo, F. Mantz, A. Rutle, and J. de Lara. A declarative and bidirectional model transformation approach based on graph co-spans. In *PPDP*, pages 1–12, 2013.

[24] E. Legros, C. Amelunxen, F. Klar, and A. Schürr. Generic and reflective graph transformations for checking and enforcement of modeling guidelines. *J. Vis. Lang. Comput.*, 20(4):252–268, 2009.

[25] R. Machado, L. Foss, and L. Ribeiro. Aspects for graph grammars. *ECEASST*, 18, 2009.

[26] F. Orejas and L. Lambers. Symbolic attributed graphs for attributed graph transformation. *ECEASST*, 30, 2010.

[27] F. Parisi-Presicce. Transformations of graph grammars. In *TAGT*, volume 1073 of *LNCS*, pages 428–442. Springer, 1994.

[28] U. Prange, H. Ehrig, and L. Lambers. Construction and properties of adhesive and weak adhesive high-level replacement categories. *Applied Categorical Structures*, 16(3):365–388, 2008.

[29] J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Generic model transformations: Write once, reuse everywhere. In *ICMT'11*, volume 6707 of *LNCS*, pages 62–77. Springer, 2011.

[30] J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Flexible model-to-model transformation templates: An application to ATL. *Journal of Object Technology*, 11(2):4: 1–28, 2012.

[31] D. Sannella and A. Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. EATCS Monographs on theoretical computer science. Springer, 2012.

[32] A. Schürr. Specification of graph translators with triple graph grammars. In *WG'94*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.

[33] S. Sen, N. Moha, V. Mahé, O. Barais, B. Baudry, and J.-M. Jézéquel. Reusable model transformations. *Software and System Modeling*, 11(1):111–125, 2012.

[34] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley, 2008.

[35] G. Taentzer, F. Mantz, and Y. Lamo. Co-transformation of graphs and type graphs with application to model co-evolution. In *ICGT'12*, volume 7562 of *LNCS*, pages 326–340. Springer, 2012.

[36] G. Taentzer and A. Rensink. Ensuring structural constraints in graph-based models with type inheritance. In *FASE*, volume 3442 of *LNCS*, pages 64–79. Springer, 2005.

[37] UML 2.4.1. `http://www.omg.org/spec/UML/2.4.1/`.

[38] M. Völter and T. Stahl. *Model-driven software development*. Wiley, 2006.

[39] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Surviving the heterogeneity jungle with composite mapping operators. In *ICMT'10*, volume 6142 of *LNCS*, pages 260–275. Springer, 2010.

[40] M. Wimmer, A. Kusel, J. Schönböck, W. Retschitzegger, W. Schwinger, and G. Kappel. On using inplace transformations for model co-evolution. In *Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL 2010)*. INRIA & Ecole des Mines de Nantes, 2010.

Appendix

This appendix provides some proofs of the results in the paper.

Theorem 1 (Compatibility of amalgamation). *Given a covering COV_E and a forward-compatible binding $B_{C,MM}$, the resulting amalgamation $A_{CI} \overset{u_c}{\rightarrow} A_S \overset{u_m}{\leftarrow} A_M$ is forward-compatible with the adapters in $B_{C,MM}$.*

Proof. (Sketch) What we need to proof is that the amalgamation of two or more adapters yields a forward-compatible adapter with the adapters involved in the amalgamation. This means that pullbacks need to be preserved by colimits. For simplicity, we provide the proof for two adapters, which can be generalized as in the proof we use pushouts, and hence the needed colimits can be constructed by iterated pushouts.

Let $CI_i \overset{c_i}{\rightarrow} S_i \overset{m_i}{\leftarrow} M_i$, $CI_j \overset{c_j}{\rightarrow} S_j \overset{m_j}{\leftarrow} M_j$ be two compatible adapters, and let $c_{ij}\colon CI_i \rightarrow CI_j$, $s_{ij}\colon S_i \rightarrow S_j$, $m_{ij}\colon M_i \rightarrow M_j$ be three M-morphisms such that the resulting squares are pullbacks, as required by the compatibility condition. The resulting amalgamation adapter $A_{CI} \overset{u_c}{\rightarrow} A_S \overset{u_m}{\leftarrow} A_M$ is compatible if squares (1) and (2) in Figure 36 are pullbacks (we omit the squares between the store and the model for simplicity, but they are also required to be pullbacks, and the proof is symmetric).

Figure 37 shows the van Kampen square resulting from the amalgamation of the two adapters, where the top and bottom faces are pushouts. These pushouts are equivalent to the co-limits of

$$\begin{tikzcd}
CI_j \arrow[r, "c_j"] \arrow[dd, bend right=60] & S_j \arrow[dd, bend left=60] \\
CI_i \arrow[u, tail, "c_{ij}"] \arrow[r, "c_i"] \arrow[d] \arrow[ur, phantom, "(P.B.)"] & S_i \arrow[u, tail, "s_{ij}"'] \arrow[d] \\
A_{CI} \arrow[r, "u_c"] \arrow[ur, phantom, "(1)"] & A_S
\end{tikzcd}
\qquad
\begin{tikzcd}
A_{CI} \arrow[r, "u_c"] & A_S \\
CI_j \arrow[u] \arrow[r, "c_j"] \arrow[ur, phantom, "(2)"] & S_j \arrow[u] \\
CI_i \arrow[u, tail, "c_{ij}"] \arrow[r, "c_i"] \arrow[uu, bend left=60] \arrow[ur, phantom, "(P.B.)"] & S_i \arrow[u, tail, "s_{ij}"'] \arrow[uu, bend right=60]
\end{tikzcd}$$

Figure 36: Conditions for compatibility of amalgamation.

$c_{ij}\colon CI_i \rightarrow CI_j$ and $s_{ij}\colon S_i \rightarrow S_j$. Then, by the compatibility of the two adapters, the back face is a pullback, and the left face is a pullback by construction. Hence, the front and right faces are pullbacks as required.

$$\begin{tikzcd}
CI_i \arrow[rr, "c_{ij}"] \arrow[dr, "id"] \arrow[dd, "c_i"'] & & CI_j \arrow[dr] \arrow[dd] & \\
& CI_i \arrow[rr] \arrow[dd, "c_i"] & & A_C \arrow[dd] \\
S_i \arrow[rr, "s_{ij}"] \arrow[dr, "id"] & & S_j \arrow[dr] & \\
& S_i \arrow[rr] & & A_S
\end{tikzcd}$$

Figure 37: Van Kampen cube derived from the amalgamation.

□

Lemma 1(Match equivalence) *Given an integrated rule* $\langle L_C \rightarrow L_S \leftarrow L_M\rangle \overset{\langle l_C,l_S,l_M\rangle}{\longleftarrow} \langle K_C \rightarrow K_S \leftarrow K_M\rangle \overset{\langle r_C,r_S,r_M\rangle}{\longrightarrow} \langle R_C \rightarrow R_S \leftarrow R_M\rangle$*, and a cospan* $E \rightarrow S \leftarrow M$ *resulting from adapting* M*, the following statements are equivalent:*
(i) $\exists m_m\colon L_M \rightarrow M$*, (ii)* $\exists m_s\colon L_S \rightarrow S$*, (iii)* $\exists m_c\colon L_C \rightarrow E$*.*

Proof. (Sketch) The diagram in Figure 38 shows the amalgamation to adapt the model (front) and the rules (back). The front and back squares are pushouts. The upper squares are pullbacks, because $C_L \rightarrow S_L \leftarrow M_L$ and $A_{CI} \rightarrow A_S \leftarrow A_M$ have been amalgamated using the same initial set of adapters, and hence are compatible. This proposition shows that both adaptations are compatible.
(i) implies (ii) and (iii). Given a M-morphism $m_m\colon L_M \rightarrow M$, we need to show the existence of $m_s\colon L_S \rightarrow S$ and $m_c\colon L_C \rightarrow E$. The front ($A_M$, A_S, S, M) and back (M_L, S_L, L_S, L_M) squares in the right cube of Figure 38 are pushouts. Hence, by the universal pushout property, there is a unique $m_s\colon L_S \rightarrow S$. The right cube is van Kampen, because the adapters $C_L \rightarrow S_L \leftarrow M_L$ and $A_{CI} \rightarrow A_S \leftarrow A_M$ are compatible, and so the top square M_L, S_L, A_S, A_M is a pullback. Finally, the right square M_L, A_M, L_M, M is also pullback (both $M_L \rightarrow L_M$ and $A_M \rightarrow M$ are epimorphisms as we have complete coverings), and hence the square S_L, A_S, L_S, S is also a

$$\begin{tikzcd}
 & C_L \arrow[dl, "c"'] \arrow[rr] \arrow[dd] & & S_L \arrow[dl, "s"'] \arrow[dd] & & M_L \arrow[ll] \arrow[dl, "m"'] \arrow[dd] & \\
A_{CI} \arrow[rr] \arrow[dd] & & A_S \arrow[dd] & & A_M \arrow[ll] \arrow[dd] & & \\
 & L_C \arrow[rr] & & L_S \arrow[dl, dotted, "m_s"'] & & L_M \arrow[ll] \arrow[dl, "m_m"'] & \\
E \arrow[rr] & & S & & M \arrow[ll] \arrow[rr, hook] & & M'
\end{tikzcd}$$

Figure 38: Match equivalence, assuming the existence of m_m.

pullback. The idea now is to construct a van Kampen cube on the left to show the existence of an M-morphism $m_c\colon L_C \to E$.

Let L'_C be the pullback object of $L_S \to S \leftarrow E$, as shown to the left of Figure 39. We will show that $L_C \cong L'_C$. By the pullback universal property, there is a unique $u\colon C_L \to L'_C$. This new cube is van Kampen, and hence the back square C_L, S_L, L_S, L'_C is a pushout. By the uniqueness of pushout complements along monomorphisms, we have that $L'_C \cong L_C$ and hence that there is $m_c\colon L_C \to E$.

$$\begin{tikzcd}
 & C_L \arrow[dl] \arrow[rr] \arrow[dd] \arrow[ddd, dotted, bend right, "u"'] & & S_L \arrow[dl] \arrow[dd] \\
A_{CI} \arrow[rr] \arrow[ddd] & & A_S \arrow[ddd] & \\
 & L_C \arrow[rr] & & L_S \arrow[ddl, "m_s"] \\
 & L'_C \arrow[urr] \arrow[dl, "m_c"'] \arrow[r, phantom, "P.B"] & & \\
E \arrow[rr] & & S &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & L_M \arrow[dl] \arrow[ddd, "m_m"] & & & K_M \arrow[lll, "l_M"'] \arrow[dl] \arrow[ddd, "m_k"] \\
 & L_S \arrow[ddd, "m_s"] & & & K_S \arrow[lll, "l_S"'] \arrow[ddd, "s_k"] & \\
L_C \arrow[ur] \arrow[ddd, "m_c"] & & & K_C \arrow[lll, "l_C"'] \arrow[ur] \arrow[ddd, "c_k"] & & \\
 & & M \arrow[dl] & & & M_D \arrow[lll, "d_m"'] \arrow[dl] \\
 & S & & & S_D \arrow[lll, "d_s"'] & \\
E \arrow[ur] & & & E_D \arrow[lll, "d_e"'] \arrow[ur] & &
\end{tikzcd}$$

Figure 39: Finding morphism $m'_c\colon L_C \to E$ (left) and applicability equivalence (right).

(iii) implies (ii) and (i). If there is a match $m_c\colon L_C \to E$ the reasoning is symmetric to the previous case.

(ii) implies (i) and (iii). Given $m_s\colon L_S \to S$, we have that the square S_L, A_S, L_S, S is a pullback (see Figure 38). Thus, we construct $m_c\colon L_C \to E$ as shown to the left of Figure 39, and then $m_m\colon L_M \to M$ in a symmetric way.

□

Lemma 2(Applicability equivalence) *Given an integrated rule $\langle L_C \to L_S \leftarrow L_M\rangle \overset{\langle l_C,l_S,l_M\rangle}{\longleftarrow} \langle K_C \to K_S \leftarrow K_M\rangle \overset{\langle r_C,r_S,r_M\rangle}{\longrightarrow} \langle R_C \to R_S \leftarrow R_M\rangle$ derived from a binding with no invalid adapters, a safe model M and a match $m_m\colon L_M \to M$, the following statements are equivalent:*

(i) rule $L_M \leftarrow K_M \to R_M$ is applicable to M' at m_m,

(ii) rule $L_M \leftarrow K_M \to R_M$ is applicable to $M \hookrightarrow M'$ at m_m,

(iii) rule $L_S \leftarrow K_S \rightarrow R_S$ is applicable at m_s (the equivalent match of m_m for model S)

(iv) rule $L_C \leftarrow K_C \rightarrow R_C$ is applicable at m_c (the equivalent match of m_m for model E)

Proof. We first check that (i) and (ii) are equivalent. First $m_m \colon L_M \rightarrow M$ exists if $m_m \colon L_M \rightarrow M'$ exists, because $M' \setminus M$ contains elements not present in the amalgamation. Second, no dangling edges can occurr in M' but not in M, because we do not allow new edges in M' from nodes present in M, because M is safe.

Now that we have shown equivalence of (i) and (ii), we check equivalence between (ii), (iii) and (iv).
(iii) implies (ii) and (iv). In the right of Figure 39, assume there exists S_D, and hence the square L_S, S, K_S, S_D is a pushout. Then, we can construct the front and back van Kampen cubes as follows. We build the pullback of $E \rightarrow S \overset{d_s}{\leftarrow} S_D$, leading to $E \overset{d_e}{\leftarrow} E_D \rightarrow S_D$. By the universal pullback property, we have that $\exists_1 c_k \colon K_C \rightarrow E_D$. As we have that the square L_C, L_S, E, S is a pullback, by the van Kampen property the front face is a pushout and hence E_D is the sought pushout complement. Symmetrically, we construct M_D as a pullback object of $M \rightarrow S \overset{d_s}{\leftarrow} S_D$. By the same reasoning, we obtain that the back face is a pushout and hence M_D is the sought pushout complement.
(ii) implies (iii). In this case, we cannot construct a van Kampen square because we would be starting by the top face pushout, reversing the implication [11]. However, we know that M is type safe and there are no invalid adapters. Therefore, there are no unsafe or external edges from MM, and so there are no unsafe or external edges in S either, and hence S is type safe. This means that if the rule is applicable at M, so is at S, and conversely, if there are dangling edges at M, so there will be at S.
(iv) implies (iii). Again, in this case we cannot construct a van Kampen square, as we would be starting by the top face pushout, reversing the implication [11]. However, if M is type safe, so it is S, and hence if the rule is applicable at E, so it will be at S, because at S we cannot have dangling unsafe edges.

□

Lemma 3(Application equivalence) *Given an integrated rule $\langle L_C \rightarrow L_S \leftarrow L_M \rangle \overset{\langle l_C, l_S, l_M \rangle}{\longleftarrow} \langle K_C \rightarrow K_S \leftarrow K_M \rangle \overset{\langle r_C, r_S, r_M \rangle}{\longrightarrow} \langle R_C \rightarrow R_S \leftarrow R_M \rangle$ derived from a binding without invalid adapters, a safe model M, a match $m_m \colon L_M \rightarrow M$, and three equivalent matches $m_m \colon L_M \rightarrow M$, $m_s \colon L_S \rightarrow S$ and $m_c \colon L_C \rightarrow S$ (see Figure 40) where:*

(i) rule $L_M \leftarrow K_M \rightarrow R_M$ is applied to M' at m_m yielding graph M''',

(ii) rule $L_M \leftarrow K_M \rightarrow R_M$ is applied to $M \hookrightarrow M'$ at m_m yielding graph $M'' \hookrightarrow M'''$,

(iii) rule $L_S \leftarrow K_S \rightarrow R_S$ is applied to S at m_s yielding graph S''

(iv) rule $L_C \leftarrow K_C \rightarrow R_C$ is applied to E at m_c yielding graph E''

then, $\exists_1 m_d \colon M_D \rightarrow S_D$, $\exists_1 s_d \colon E_D \rightarrow S_D$, $\exists_1 m' \colon M'' \rightarrow S''$, $\exists_1 s' \colon E'' \rightarrow S''$ s.t. the bottom squares in Figure 34 are pullbacks.

Proof. Consider the cube to the left of Figure 41. The front and back faces are pushouts (first step in rule applications), and the top is a pullback by rule construction. To find $m_d \colon M_D \rightarrow S_D$, we calculate the pullback of $S_D \rightarrow S \leftarrow M$. There is a unique arrow to the pullback object

$$\begin{tikzcd}
 & & L_M \arrow[dl] \arrow[ddd, "m_m"] & & & K_M \arrow[lll, "l_M"'] \arrow[rrr, "r_M"] \arrow[dl] \arrow[ddd] & & & R_M \arrow[dl] \arrow[ddd] \\
 & L_S \arrow[ddd, "m_s"] & & & K_S \arrow[lll, "l_S"'] \arrow[rrr, "r_S"] \arrow[ddd] & & & R_S \arrow[ddd] & \\
L_C \arrow[ur] \arrow[ddd, "m_c"] & & & K_C \arrow[lll, "l_C"'] \arrow[rrr, "r_C"] \arrow[ur] \arrow[ddd] & & & R_C \arrow[ur] \arrow[ddd] & & \\
 & & M \arrow[dl] & & & M_D \arrow[lll] \arrow[rrr] \arrow[dl, dotted, "m_d"] & & & M'' \arrow[dl, dotted, "m'"] \\
 & S & & & S_D \arrow[lll] \arrow[rrr] & & & S'' & \\
E' \arrow[ur] & & & E_D \arrow[lll] \arrow[rrr] \arrow[ur, dotted, "s_d"] & & & E'' \arrow[ur, dotted, "s'"] & &
\end{tikzcd}$$

Figure 40: Transforming M, S and E, and obtaining $E'' \to S'' \leftarrow M''$.

$u\colon K_M \to M'_D$. This new cube is van Kampen. Hence, by uniqueness of the pushout complement, we obtain that $M'_D \cong M_D$ and $m_d\colon M_D \to S_D$, being m_d the unique arrow making the square a pullback. Then, by the pushout universal property, $\exists_1 m'\colon M'' \to S''$ (see the right of Figure 41, where the back face is a pushout).

$$\begin{tikzcd}
 & K_M \arrow[rr] \arrow[dl] \arrow[dd] \arrow[ddd, dotted, bend right, "u"'] & & L_M \arrow[dl] \arrow[dd] \\
K_S \arrow[rr] \arrow[dddd] & & L_S \arrow[dddd] & \\
 & M_D \arrow[rr] & & M \arrow[ddl] \\
 & M'_D \arrow[urr] \arrow[dl, "m_d"'] \arrow[dr, phantom, "P.B"] & & \\
S_D \arrow[rr] & & S &
\end{tikzcd}
\qquad
\begin{tikzcd}
K_M \arrow[rr] \arrow[dr] \arrow[dd] & & R_M \arrow[dr] \arrow[dd] & \\
 & K_S \arrow[rr] \arrow[dd] & & R_S \arrow[dd] \\
M_D \arrow[rr] \arrow[dr, "m_d"'] & & M'' \arrow[dr, dotted, "m'"] & \\
 & S_D \arrow[rr] & & S''
\end{tikzcd}$$

Figure 41: Finding morphism $m_d\colon M_D \to S_D$ (left). Finding morphism $m'\colon M'' \to S''$.

A similar reasoning can be used to obtain $s_d\colon E_D \to S_D$ and $s'\colon E'' \to S''$.

□