Reusable Graph Transformation Templates

Juan de Lara and Esther Guerra

Department of Computer Science
Universidad Autónoma de Madrid (Spain)
{Juan.deLara,Esther.Guerra}@uam.es

Abstract. Model-Driven Engineering promotes models as the principal artefacts of the development, hence model transformation techniques – like graph transformation – become key enablers for this development paradigm. In order to increase the adoption of Model-Driven Engineering in industrial practice, techniques aimed at raising the quality and productivity in model transformation development are needed.
In this paper we bring elements from *generic programming* into graph transformation in order to define generic graph transformations that can be reused in different contexts. In particular, we propose the definition and instantiation of graph transformation *templates* whose requirements from generic types are specified through so-called *concepts*, as well as *mixin layers* that extend meta-models with the extra auxiliary elements needed by templates.

1 Introduction

Model-Driven Engineering (MDE) is a software development paradigm that promotes the use of models as the principal assets of the development. Hence, model manipulation techniques – like graph transformation (GT) [4] – become enabling technologies for this approach.

In order to foster the use of MDE in industry, techniques aimed at raising the quality of the generated software and to speed up the productivity of engineers are needed. One way to improve productivity is to increase the *reusability* of model transformations, so that they can be applied in different contexts. Unfortunately, building transformations in MDE is a type-centric activity, in the sense that transformations are defined over the specific types of concrete meta-models, and it is difficult to reuse them for other meta-models even if they share characteristics. For example, even if many languages share the semantics of Petri nets, like activity diagrams or process modelling languages, such semantics is normally defined over some specific meta-model (a concrete realization of a Petri net meta-model) and cannot be easily reused for other meta-models.

The present work aims at providing mechanisms to enable the *correct* reuse of GT systems across different meta-models. For this purpose, we build upon some ideas from generic programming [7] to define generic GT systems that we call GT *templates*. These generic GT systems are not defined over the types of concrete meta-models, but over variable types that need to be bound to types

of some specific meta-model. However, not every meta-model qualifies as a valid binding for the variable types used in a GT template. Hence, in order to ensure a *correct* reuse, we specify the requirements that meta-models need to satisfy using a so-called *concept* [3, 8]. A concept gathers the structural requirements that need to be found in a meta-model to be able to instantiate a GT template on the meta-model types and apply the template to the meta-model instances.

In addition, GTs sometimes need auxiliary model elements to perform some computations. For example, in order to define the semantics of Petri nets, we may need an edge referencing the transition that is currently being fired, or in object-oriented systems we may need to introduce an auxiliary edge to flatten the inheritance hierarchy. These extra elements do not belong to the meta-model of the language, but are auxiliary devices needed by the transformation. Hence, a GT template cannot demand specific meta-models to include such extra devices as part of its requirements. Instead, we define so-called *mixin layers* [3, 15]. These are meta-models with parameters, that are *applied* to specific meta-models, increasing them with extra elements by a gluing construction. Mixins are generic, and hence applicable to any meta-model that satisfies the requirements given by a set of concepts. In this way, a GT template can be defined over the types of the mixin and the types of the concepts such mixin needs.

This paper continues our research on model transformation reuse by means of genericity [3, 13]. While in [13] we added genericity to model-to-model transformations expressed in the ATL language, here we focus on in-place transformations expressed using GT. The use of a formal framework helps in formulating template instantiations (i.e. bindings) precisely, identifying the needed conditions for correct template reuse, and understanding the composition mechanism of mixins. The formal semantics of DPO graph transformation yields tighter conditions for correct template reuse than we obtained in previous works [3, 13].
Paper organization. First, Section 2 introduces our approach. Then, Section 3 defines meta-models algebraically. Section 4 explains how to bind concepts to meta-models, providing a mechanism to instantiate GT templates. Section 5 shows how to define and apply mixins. Section 6 provides further examples. Finally, Section 7 compares with related work and Section 8 concludes.

2 Overview of the Approach

Frequently, very similar transformations are developed for different meta-models. The reason is that although similar, each transformation is developed to work with the types of a particular meta-model, so that its use with types of other unrelated meta-models is not possible. This results in a waste of effort as the same problems and solutions have to be tackled repeatedly. For example, there is a catalogue of well-known refactorings for object-oriented systems [6]; however, if we encode them as GT rules, we need a different encoding for each meta-model we use. In this way, we need to encode a different transformation for the UML meta-model, the Java meta-model, or the meta-model of any other object-oriented notation we may like to work with.

Analogously, there are languages with similar semantics. For example, many languages share the semantics of Petri nets, such as activity diagrams or domain-specific languages for manufacturing (parts are produced and consumed at machines) and networking (packets are sent and received by computers). However, if we specify their semantics through a GT system, we need to encode a different system for the meta-model of each language.

Therefore, a mechanism to define GT systems for families of meta-models sharing some requirements would promote the reutilization of transformations. For this purpose, we use so-called *concepts* [3, 8] to gather the requirements of a family of meta-models, needed by a reusable GT system to work. These requirements are structural, and hence a concept has the form of a meta-model as well. However its elements (nodes, edges, attributes) are interpreted as variables to be *bound* to elements of specific meta-models. The rules of a *GT template* use the variables in the concept instead of the types of a specific meta-model. As a result we obtain reusability because the concept can be bound to a family of meta-models, and the GT template becomes applicable to any of them.

This situation is illustrated in Fig. 1, where a GT template has been defined over the variable types of a concept C. As an example, C may define the core structural elements that characterize Petri net-like languages, and the transformation may include rules (defined over the type variables in C) to refactor Petri net-like models, according to the catalogue in [12]. The concept can be bound to a set $L(C)$ of meta-models sharing the structure required by the concept. In this way, the rules can be applied on instances of any meta-model $MM \in L(C)$. We have depicted the set of models conformant to a meta-model MM as $L(MM)$.

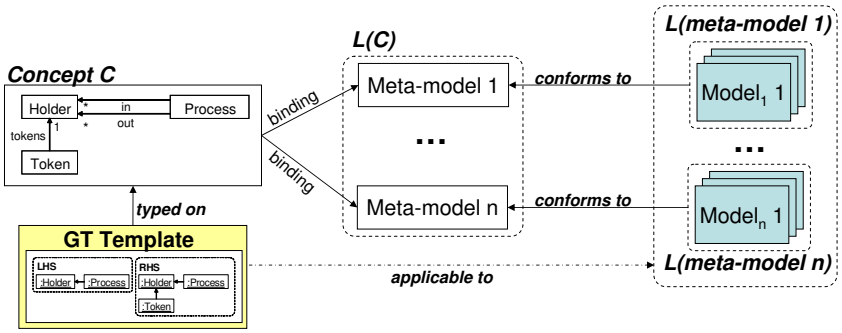


Fig. 1. Scheme of the approach: concept, GT template and binding.

In addition, oftentimes, GTs make use of auxiliary graph elements to implement some model manipulations. For example, when defining the semantics of Petri nets, we need an auxiliary node to mark which transition is being fired, as well as the processed input and output places, in order to add/remove tokens to/from the appropriate places. The types of these auxiliary elements do not belong to the meta-model of the language (Petri nets), but are auxiliary elements

needed just for the simulation. If this simulator is built as a GT template over a concept, then this concept cannot include the auxiliary elements (and no binding will be provided for them) because the meta-models will hardly ever include such extra devices.

In order to solve this problem, we define so-called *mixin layers* as an extension mechanism for meta-models. A *mixin* is a generic meta-model defining all extra elements needed by a GT template but which are not present in a concept. In addition, it includes some formal parameters acting as gluing points between the mixin and the concept. Thus, the GT template can use the variable types defined on both the mixin and the concept. Once we bind the concept into a specific meta-model, this is extended with the new types defined by the mixin, and the template can be applied on instances of this extended meta-model.

This scheme is shown in Fig. 2. In particular, the mixin adds some auxiliary elements to simulate Petri net-like languages (a pointer to the process being fired and to the processed holders of tokens). The mixin defines as gluing points the nodes `Holder` and `Process`, whose requirements are given by a concept C (structure of Petri net-like languages). Binding the concept to a specific meta-model will increase the meta-model with the elements of the mixin. The GT template is defined over the types resulting from gluing C and the mixin, and is applicable to instances of meta-models to which we can bind C.

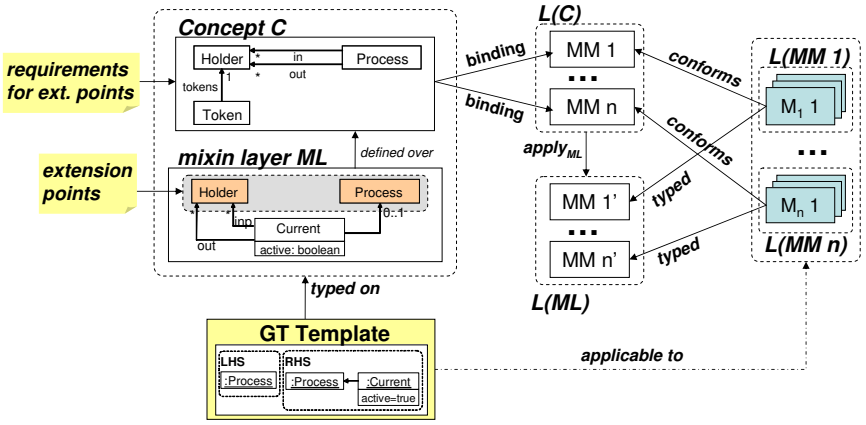


Fig. 2. Scheme of the approach: adding auxiliary elements through a mixin layer.

3 An Algebraic Setting for Models and Meta-Models

In our setting, we consider attributed[1], typed graphs [4] of the form $G = \langle V; E; A; D; src_E, tar_E \colon E \to V; src_A \colon A \to V; tar_A \colon A \to D \rangle$, made of a set V of

[1] For simplicity, we do not consider abstract nodes or attributes in edges.

vertices, a set E of edges, a set A of attributes, a set D of data nodes, and functions *src* and *tar* that return the source and target vertices of an edge, and the owning vertex and data value of an attribute.

In order to represent meta-models, we consider type graphs with inheritance and with cardinality constraints in associations, in the style of [16]. In this way, a meta-model $MM = \langle G; I \subseteq V \times V; card\colon E \to \mathbb{N} \times (\mathbb{N} \cup \{*\}) \rangle$ is made of a graph G, together with a set I of inheritance relations, and a function *card* that returns the cardinality of target ends of edges. We assume EMF-like references for edges (i.e. with cardinality only in the target end) so that UML-like associations (i.e. with cardinality in both ends) should be modelled as a pair of edges in both directions. Given a node $n \in V$, we define its clan as the set of its direct and indirect children, including itself. Formally, $clan(n) = \{n' \in V | (n', n) \in I^*\}$, where I^* is the reflexive and transitive closure of I. For simplicity, we avoid adding an algebra to MM.

Similar to [16], we give semantics to cardinality constraints by means of positive and negative atomic graph constraints [4] of the form $PC(a\colon P \to Q)$ and $\neg PC(a\colon P \to Q)$. The former require that for each occurrence of P in a graph G, we find a commuting occurrence of Q. Formally, for each $m\colon P \to G$ we need $m'\colon Q \to G$ s.t. $m = m' \circ a$. Negative atomic constraints demand that for each occurrence of P in a graph, there is no commuting occurrence of Q. If G satisfies a constraint a, we write $G \models a$.

In particular, we generate graph constraints regulating the minimum and maximum number of instances at association ends. Thus, for each edge e in the meta-model with $card(e) = [l, h]$, if $l > 0$ we generate a positive graph constraint as shown to the left of Fig. 3, and if $h \neq *$ we generate a negative graph constraint as shown to the right of the same figure.

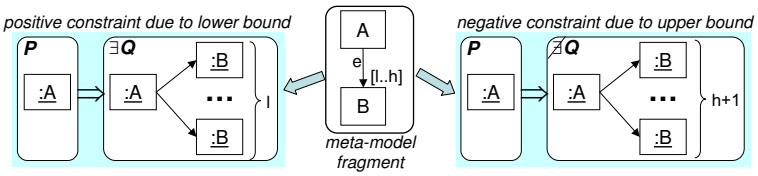


Fig. 3. Graph constraints generated from cardinality constraints.

Next we define morphisms between meta-models as a graph morphism with some extra constraints given by the inheritance hierarchy. We will use this notion later to define the binding between a concept and a meta-model.

A morphism $MM \to MM'$ between two meta-models (named MM-morphism) is given by a clan morphism [2] $f\colon G_{MM} \to MM'$ from the graph G_{MM} of the first meta-model to the second meta-model, preserving the inheritance hierarchy. A clan morphism is similar to a standard E-Graph morphism [4], but it also takes into account the semantics of inheritance. Hence, for each edge e of G_{MM} that is mapped to an edge e' of $G_{MM'}$, we allow the source node

of e to be mapped to any node in the clan of the source node of e'. Formally, $f_V(src_E(e)) \in clan(src'_E(f_E(e)))$, 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, $f_A(src_A(a)) \in clan(src'_A(f_A(a)))$. As in [9], the morphism has to preserve the inheritance hierarchy as well, hence if $(u, v) \in I_{MM}$, then $(f(u), f(v)) \in I^*_{MM'}$. Please note that we purposely neglect cardinality constraints in MM-morphisms because the semantics of these is given by graph constraints. We will deal with this issue when defining the binding between a concept and a meta-model.

A model M can be seen as a meta-model with empty inheritance hierarchy and no cardinality constraints. Therefore, we can represent the typing function $M \xrightarrow{type} MM$ as an MM-morphism. In addition, we say that M *conforms to* MM (written $M \models^{type} MM$) if there is a typing $M \xrightarrow{type} MM$ and M satisfies all graph constraints derived from the cardinality constraints in MM.

Example. Fig. 4 shows an MM-morphism f between two meta-models, where we have represented attributes as arrows to a datatype (see `a`) and mapped elements with primas (e.g. node `A` is mapped to node `A'`). The attribute in node `B` is mapped to an attribute defined in `E`, which is a supertype of the node mapped to `B`, and the same for the edge `e`. Regarding the preservation of the inheritance hierarchy, MM-morphisms permit introducing intermediate nodes in the hierarchy of the target meta-model (like node `E` which is between the image nodes `A'` and `B'`) as well as mapping several nodes in an inheritance relation into a single node.

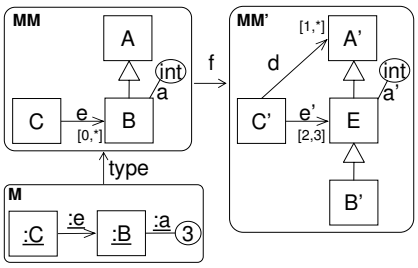


Fig. 4. MM-morphisms.

The figure also shows a typing MM-morphism $type\colon M \to MM$ using the UML notation for typing. We can compose MM-morphisms, hence M is also typed by $f \circ type\colon M \to MM'$. However, *conformance* is not compositional in general, as M conforms to MM ($M \models^{type} MM$) but not to MM' due to its cardinality constraints ($M \not\models^{f \circ type} MM'$). Finally, given $M' \models^{type} MM'$, we have that the pullback object of $MM \to MM' \leftarrow M'$ is typed by MM, but need not be conformant to MM.

4 Graph Transformation Templates

GT templates are standard GT systems specified over the variable types of a concept, which has the form of a meta-model. As an example, Fig. 5 shows to the left the concept *TokenHolder*, which describes the structural requirements that we ask from Petri net-like languages, namely the existence of classes playing the roles of token, holder (places in Petri nets) and process (transitions). We can use this concept to define generic GT systems for the simulation and refactoring of models in languages with this semantics. For instance, the right of the same

figure shows one of the behaviour-preserving refactoring rules proposed in [12] expressed in a generic way, using the types of the concept. The rule removes self-loop holders with one token provided they are connected to a single process.

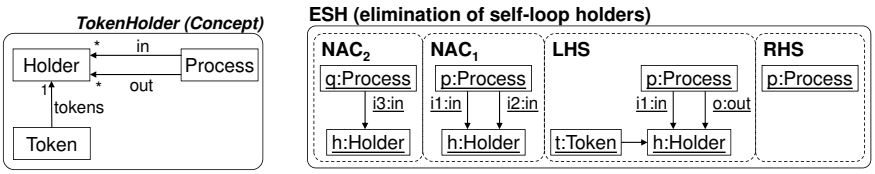


Fig. 5. Concept *TokenHolder* (left). GT template over the concept (right).

In order to use the GT template, we need to *bind* the concept to a specific meta-model. This binding is an MM-morphism with some extra constraints derived from the particular GT template to be reused. As an example, Fig. 6 shows a binding attempt from the *TokenHolder* concept to a meta-model to define factories. *Factory* models contain machines interconnected by conveyors which may carry parts. Both conveyors and machines need to be attended by operators. Hence, our aim is to apply the rule template *ESH* on instances of the *Factories* meta-model. However, there is a problem because this rule deletes holders, and we have bound holders to conveyors, which are always connected to some operator as required by the cardinality constraints. Thus, if we try to apply the rule (using DPO semantics), it will always fail as we will obtain a dangling edge. This shows that the binding should ensure some correctness conditions, which we present in the remaining of the section.

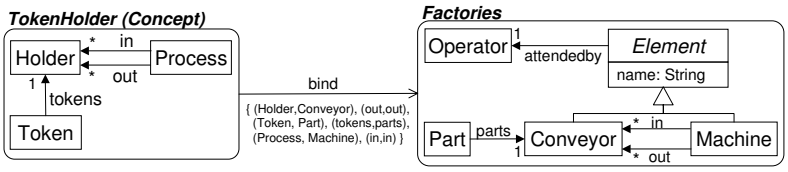
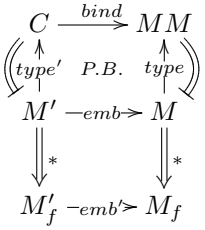


Fig. 6. Binding a concept to a meta-model: first attempt.

The general instantiation scheme of a GT template for a given binding is shown in the diagram to the right. The template is typed over the concept C, which needs to be bound to a meta-model MM by a special kind of MM-morphism $bind\colon C \to MM$. The binding provides a re-typing for the rules of the template, which then become applicable to instance graphs of MM. In order to ensure a correct application of the template, we must guarantee that for every model



$M \models^{type} MM$, if we consider only the elements M' given by the concept C (obtained by the pullback in the diagram), and $M' \models C$, then, for each possible sequence of rule applications over M', there is a sequence of rule applications (using the same order) applicable to M. Moreover, there should be an embedding from the final model M'_f into M_f.

Although this issue resembles the one handled by the embedding and extension theorems [4], there are fundamental differences. The main one is that our goal is to discard invalid bindings and initial models that would lead to an incorrect application of the GT template *before* applying it. On the contrary, the mentioned theorems check the feasibility of each particular derivation $M' \Rightarrow^* M'_f$ checking some conditions *after* the transformation is performed. Hence, our view gains in efficiency. Moreover, the embedding and extension theorems do not consider inheritance and cardinalities in type graphs.

In order to ensure that a GT template will behave as expected for a given meta-model, we generate two kinds of constraints. The first kind works at the meta-model level and forbids bindings that would *always* lead to an incorrect execution of the template, as some rules of the template would be inapplicable for any possible model due to dangling edges or violations of cardinality constraints. However, some bindings may lead to incorrect executions only for *some* initial models. Thus, our second kind of constraints detects potential incorrect template executions for a particular instance of a meta-model. If a binding and an initial model satisfy these constraints, then the template can be safely applied to the model. Next we explain in detail each constraint type.

Constraints for bindings. These constraints act as application conditions for the *binding*. Fig. 7(a) illustrates how they work. Assume we want to apply our refactoring GT template to models that conform to the *Factory* meta-model. The first step is therefore binding the *TokenHolder* concept to the meta-model, as indicated in the figure. Looking at the rule in Fig. 5 we notice that it deletes holders. As holders are mapped to conveyors, and conveyors are always attended by one operator, applying the rule to a model with a conveyor will always produce a dangling edge making the rule not applicable. Since this is not the behaviour expected from the original template, this binding is not allowed. In order to detect these situations, we attach the atomic constraint $\neg PC(TokenHolder \xrightarrow{q} MandatoryEdge)$ shown in the figure to the binding, in a similar way as application conditions are attached to the LHS of a rule. This constraint forbids a binding if the holder is connected to some node Z through an edge with lower cardinality bigger than 0. As this is the case should we identify e and `attendedby`, the binding is not allowed.

The structure of the generated constraints for bindings is depicted in Fig. 7(b). The constraints use MM-morphisms and must be satisfied by the function *bind*. They are generated for each node type that is created or deleted by the GT template. Thus, if the template creates or deletes an object of type A, we generate a constraint that has the concept C as premise, and the concept with a class X (existing or not in C) connected to A through an edge e with lower cardinality bigger than 0 as consequence. We also demand that the match of the edge e

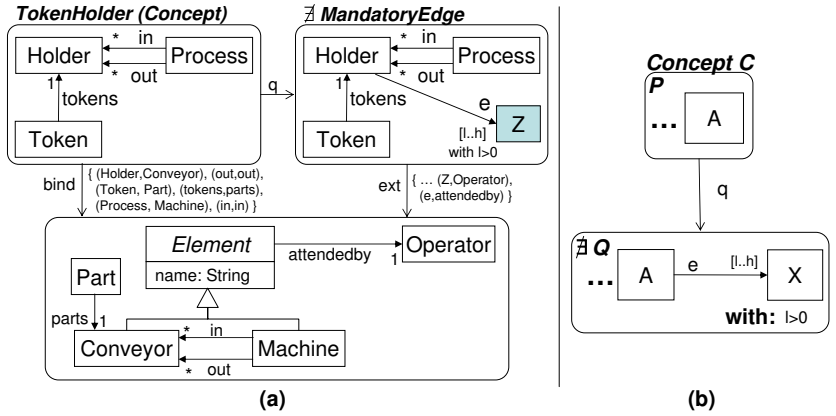


Fig. 7. (a) Evaluating constraint in a binding. (b) Scheme of constraints for bindings.

does not belong to $bind(C)$. If a rule deletes an A object, the intuitive meaning is that it will produce a dangling edge because of this mandatory edge, disabling the rule application. This breaks the correctness criteria because we could apply the rule in a model M' conformant to the concept but not in a model M conformant to the meta-model. Therefore we forbid such a binding. If a rule creates an A object, we generate the same constraint because applying the grammar to a model M typed over MM would only produce incorrect models, as the created object would need to be connected to an object of type X through an unforeseen edge type e. Please note that the generated constraint takes into account the case where the edge e has been defined in an ancestor of the class bound to A, as we use MM-morphisms. On the contrary, the constraint does not detect if there is a subtype defining an unbounded mandatory edge and does not forbid the binding in such a case. Indeed, in this situation, there may be initial models where the template can be safely applied, therefore this scenario is handled by a second set of constraints working at the model level (see below).

Regarding cardinalities, we forbid binding edges with cardinality $[l..h]$ to edges with cardinality $[l'..h']$ if both intervals do not intersect, as from a model $M \models MM$ we would never obtain a model $M' \models C$ performing the pullback. In addition, if a GT template creates or deletes instances of the source or target classes of an edge e defined in a concept C, then we can map the edge to $e' = bind(e)$ only if $card(e) = card(e')$. This condition is not required for edges whose source and target are not created or deleted by the template; however, the initial model must satisfy the cardinality constraints of the concept in any case (i.e. the pullback object M' must satisfy $M' \models C$).

Constraints for initial models. These constraints check whether a GT template can be safely executed on a given initial model. For instance, assume that the cardinality of `attendedby` is [0..1] in the meta-model of Fig. 7(a), so that the binding is allowed. Still, given an initial model like the one in Fig. 8(a), our generic refactoring will not be applied to the conveyor as it has one operator,

hence leading to a dangling edge as discussed before. This differs from the original template behaviour where such dangling edges do not occur. However, we can safely apply the rule to any model where conveyors have no operator, which we check by generating the constraint in the upper part of the figure.

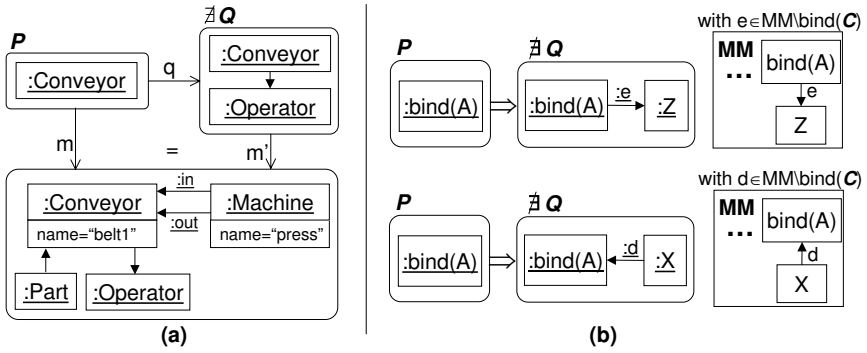


Fig. 8. (a) Evaluating constraint in model. (b) Scheme of constraints for initial models.

Fig. 8(b) shows the structure of the generated constraints for initial models. These constraints restrict the instances of the meta-model MM to which the concept C is bound. They are generated for each node type in the meta-model that is deleted by the GT template, as well as for its subtypes, whenever the types declare an edge not included in the binding. Formally, if a generic rule deletes an object of type B, we generate a constraint for each $A \in clan(bind(B))$ and for each edge e in which A participates whose type belongs to $MM \setminus bind(C)$. This is a sufficient condition to avoid violating the correctness criteria (the instantiated template could fail due to dangling edges) but it is not a necessary condition.

5 Mixin Layers

We now define mixin layers with the purpose of extending meta-models with any auxiliary element needed to execute a GT template. A mixin layer is a meta-model where some of its elements are identified as parameters. Parameters are interpreted as variable types that have to be instantiated to types of the specific meta-model where we want to apply the mixin. However, not every meta-model is eligible to be extended by a particular mixin, and not every type is a valid instantiation of the mixin parameters. The requirements needed by a meta-model and its types are given by one or more concepts.

We define a mixin layer ML as $ML = \langle MM, Conc = \{C_i\}_{i \in I}, Par = \{P_j\}_{j \in J} \rangle$, where MM is a meta-model, $Conc$ is a set of concepts expressing the requirements for meta-models to be extensible by the mixin, and Par is a set of parameters identifying the mixin extension points. Each element P_j in the set of parameters has the form $P_j = \langle G_{MM} \leftarrow G_j \rightarrow G_{C_i} \rangle$, a span relating the

graphical elements in the mixin (G_{MM}) with the graphical elements in one of the concepts ($G_{C_i \in Conc}$).

Example. We are building a generic simulator for Petri net-like languages by means of a GT template defined over the concept *TokenHolder*. However, apart from the elements already present in this concept, the simulator must use auxiliary nodes and edges to model the firing of transitions. Adding these elements to the concept is not an option because it would imply that the definition of every language to be simulated with the template should be modified manually to include these auxiliary elements in its meta-model. Instead, we define a mixin which increases any meta-model to be simulated with these auxiliary elements in a non-intrusive way. Fig. 9 shows the mixin (dotted box named "Simulation mixin") which declares two parameters (shaded classes `Holder` and `Process`). Additionally, the concept *TokenHolder* gathers the requirements for the eligible meta-models for the mixin. The relation between the mixin and the concept is expressed as a span of MM-morphisms. This is used to build the meta-model shown to the right by a gluing construction (a colimit, even though in the particular case of the figure it is also a pushout). This meta-model contains all variable types that the GT template can use.

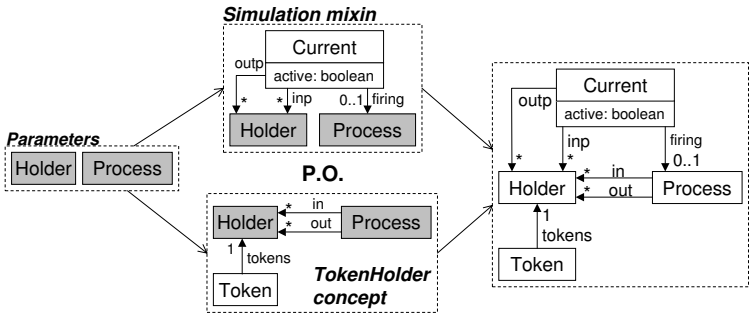


Fig. 9. Specifying a mixin for the simulation of Petri net-like languages.

Fig. 10 shows some rules of the generic simulator defined over the mixin. The rule to the left selects one enabled process to be fired, marking it with an instance of the `Current` class from the mixin. Its application condition checks the enabledness of the process (i.e. each input holder has a token). The rule to the right removes one token from an input holder of the process being fired, marking it as processed. Additional rules produce tokens in output holders, and unmark the processed holders and process to allow further firings.

A mixin becomes applicable by binding its concepts to a meta-model. Fig. 11 shows the binding of concept *TokenHolder* to the *Factory* meta-model. Then, the bound meta-model is extended with the elements defined in the mixin meta-model but not in the concept (class `Current` and edges `inp`, `outp` and `firing`). Thus, we can apply the GT template to instances of the resulting meta-model.

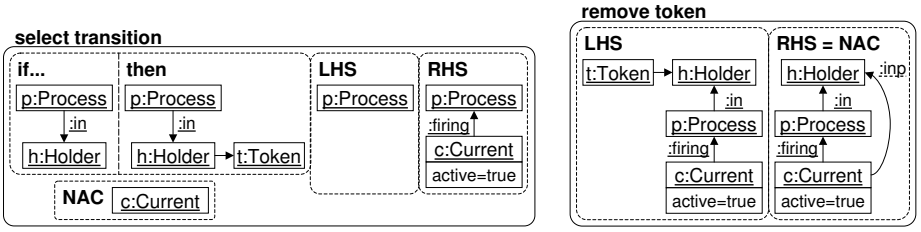


Fig. 10. Some rules of the GT template for the simulation of Petri net-like languages. The template is defined over the mixin *Simulation* shown in Fig. 9.

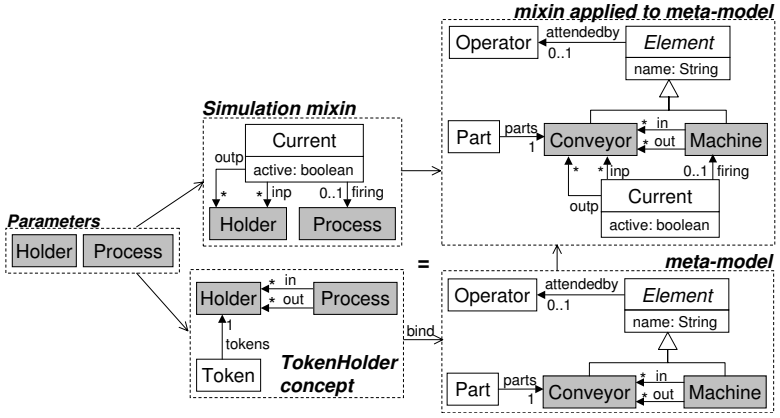


Fig. 11. Applying mixin *Simulation* to a meta-model.

The left of Fig. 12 presents formally how a mixin is applied to a meta-model. The mixin in the figure defines a meta-model MM_{ml}, a set of concepts C_i and parameters $MM_{ml} \leftarrow G_j \rightarrow C_i$. The gluing of the mixin meta-model and the concepts is obtained by calculating their colimit[2], yielding object $\overline{MM_{ml}}$. This is the meta-model over which a GT template is defined. For instance, the template rules shown in Fig. 10 use the meta-model to the right of Fig. 9. Next, the mixin can be applied by binding its concepts to a particular meta-model (or in general to a set of meta-models, as it is not necessary to bind all concepts to the same meta-model). The colimit of the different mixin parameters $MM_{ml} \leftarrow G_j \rightarrow C_i$ and the bound meta-models yields $\overline{MM}$, which is used to retype the template for its application on the bound meta-models. By the colimit universal property, there is a unique commuting $u\colon \overline{MM_{ml}} \rightarrow \overline{MM}$, which acts as binding between the meta-model over which the GT template is specified and the extended specific meta-model. The right of Fig. 12 shows this unique binding for the example.

[2] in the category of MM-objects and MM-morphisms.

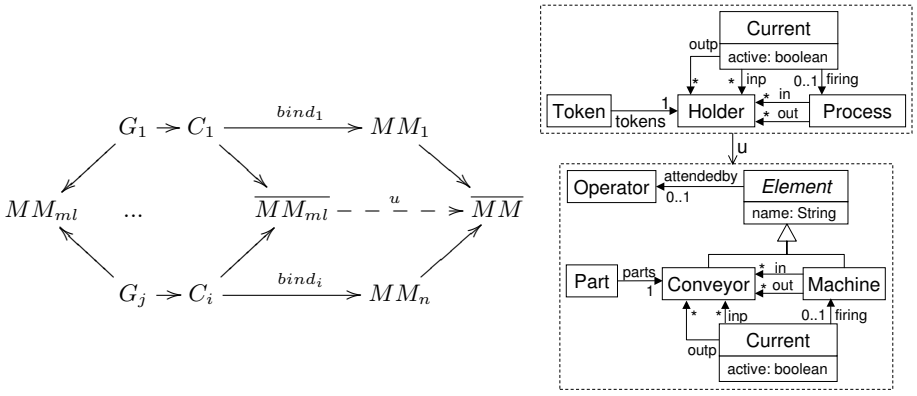


Fig. 12. Binding and applying a mixin (left). Resulting binding for the example (right).

6 Additional examples

Next we show a further example illustrating the applicability of our proposal.

Software Engineers have developed a catalog of refactorings to improve the quality of software systems without changing its functionality [6]. One of the most well known catalogs is specially tailored for object-oriented systems [6]. This catalog describes rules applicable to any object-oriented language, but their encoding for a particular object-oriented notation cannot be reused to refactor other notations. Thus, a developer should encode different rule sets for the Java meta-model, the UML meta-model, and so on.

In our approach, we can define the refactorings once over a concept and then bind the concept to several meta-models, obtaining reuse for each bound meta-model. Fig. 13 shows (a simplification of) the concept, together with bindings for two meta-models. The one to the left defines simple UML class diagrams. The right meta-model is for Rule-Based Access Control (RBAC) [14], and permits the definition of properties and permissions for roles that can be hierarchically arranged. Children roles inherit the properties and permissions of parent roles.

Fig. 14 shows the generic rules for the *pull-up attribute* refactoring [6], which moves an attribute to a superclass when all its children classes define it. The first rule detects the refactoring opportunity and moves the attribute from one of the children classes to the superclass. Then, the second rule removes the attribute from the rest of children. We can also implement a more refined version of this refactoring by defining a mixin that declares a pointer for classes, which the rules can use to indicate the parent class being refactored (i.e. class `p` in the rules).

The binding $bind_1$ in Fig. 13 allows applying the generic refactoring to UML models and pull-up fields. Nonetheless, using a different binding permits refactoring references and methods as well, mapping `Attribute` to `Method` or to `Reference`. Similarly, the binding $bind_2$ permits refactoring properties in role hierarchies, but we can bind `Attribute` to `Operation` to pull-up operations.

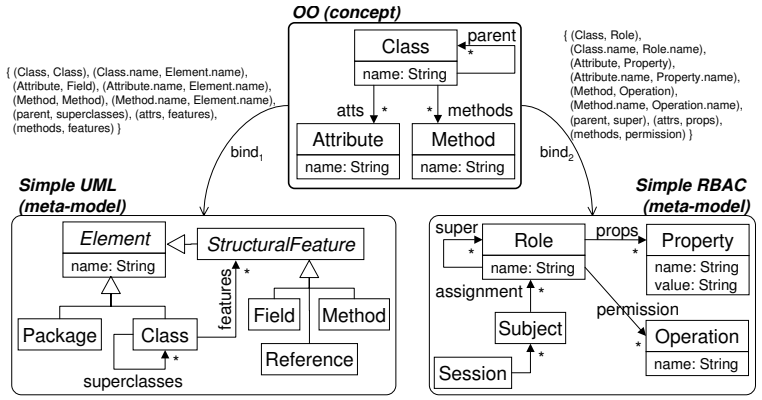


Fig. 13. Concept for Object-Oriented systems, and two possible bindings.

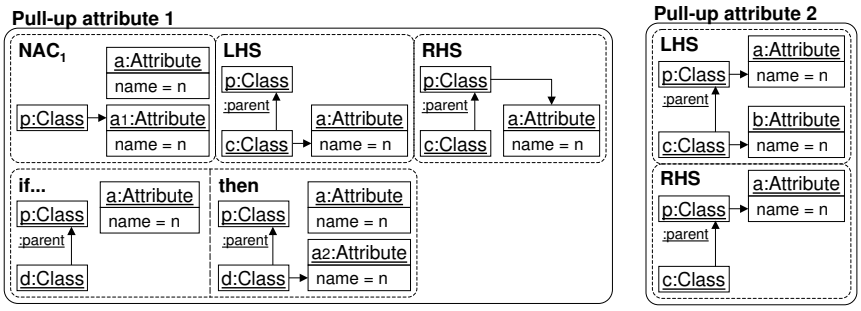


Fig. 14. Rules of the GT template implementing refactoring *pull-up attribute*.

7 Related Work

In object-oriented programming, mixins and traits are classes that provide extra functionality without being instantiated. Instead, other classes may inherit from the mixin, which is a means to collect functionality. We have generalized this idea to mixin layers. These are parameterised meta-models adding auxiliary elements to a set of meta-models sharing the characteristics specified by a concept.

In previous work [3], we brought ideas from generic programming into MDE, implementing them in the MetaDepth tool. In particular, we were able to write generic model manipulations as EOL programs, an OCL-like language with side effects. Here we have adapted these ideas to the algebraic framework of GT, which presents several advantages: (i) we were able to formulate correctness criteria for instantiation and application of GT templates; (ii) in contrast to EOL programs, GT permits analysing the effects of transformations, and this is useful to discard bindings leading to incorrect applications of the GT templates; (iii) formalizing bindings as morphisms provides a more precise description of the binding constraints, which we could not do in [3] because the behaviour of

EOL programs cannot be easily analysed; and (iv) the algebraic formalization of mixins helped us in understanding how they work. Moreover, we discovered that a pattern-based approach to genericity (like the one presented here) imposes less restrictive conditions for the binding than one based on a scripting language (like EOL or OCL). This is so as MM-morphisms allow defining the target of a reference in a supertype (cf. reference e' in Fig. 4). In EOL, a navigation expression $c.e$ may lead to an E object, which may not have all properties of a B' object (as expected by the generic operation). On the contrary, in GT one provides a pattern with an explicit type for its objects (e.g. B which gets mapped to B'), hence filtering the undesired E objects.

Parameterized modules were proposed in algebraic specification in the eighties [5]. A parameterized module is usually represented with a morphism $par\colon P \to M$ from the formal parameters to the module. In this paper, we propose using *concepts* to restrict how the formal parameters can be bound to the actual parameters in mixins. We can also think as GT templates as parameterized models (by a concept). In this case, the special semantics of DPO GT induce additional constraints in the binding. Our MM-morphisms are based on S-morphisms [9], but we support attributes and do not require morphisms to be subtype preserving. Our composition mechanism is also related to Aspect-Oriented Modelling [10], which focuses on modularizing and composing crosscutting concerns.

In the context of GT, there are some proposals for adding genericity to rules. For example, the VIATRA2 framework [1] supports generic rules where types can be rule parameters. MOFLON has also been extended with generic and reflective rules using the Java Metadata Interface [11]. These 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. Still, none of these tools provide mechanisms (like *concepts* and *bindings*) to control the correctness of rule applications, or extension mechanism (like mixins) for meta-models. We believe that the ideas presented in this paper can be adapted to these two approaches.

8 Conclusions and Future Work

In this paper we have adapted *generic programming* techniques to increase the reusability of GTs. In particular, we have defined GT templates, which are not typed over a specific meta-model, but over *concepts* specifying the structural requirements that a meta-model should fulfil if we want to apply the template on its instances. Hence, the GT template can be instantiated for any meta-model satisfying the concept, obtaining reusable transformations. Besides, many GT systems use auxiliary elements that have to be included ad-hoc in the meta-models. We have proposed a non-intrusive solution consisting on the definition of mixin layers declaring any extra device used by the template. The requirements that a meta-model should fulfil to be extended through the mixin are given by a concept. Again, we obtain reusability because a template that uses types of a mixin can be applied to any meta-model that satisfies the mixin requirements.

As for future work, currently we forbid bindings that can lead to an incorrect execution of a GT template. However, it may be sometimes possible to semi-automatically adapt the template to make it work correctly. In addition, now we require an exact match of the concept in the meta-models, but to increase reusability, we plan to provide techniques to resolve some heterogeneities in the binding, in the line of [13]. On the practical side, we plan to include the lessons learnt regarding correct binding and correct reuse in our MetaDepth tool.

Acknowledgements. Work funded by the Spanish Ministry of Science (projects TIN2008-02081 and TIN2011-24139) and the Region of Madrid (project S2009/TIC-1650).

References

1. A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *SAC'06*, pages 1280–1287, 2006.
2. J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *TCS*, 376(3):139–163, 2007.
3. J. de Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *MoDELS'10*, volume 6394 of *LNCS*, pages 16–30. Springer, 2010.
4. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer-Verlag, 2006.
5. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. Springer, Monographs in Theor. Comp. Sci., 1990.
6. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
7. R. García, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. *SIGPLAN*, 38(11):115–134, 2003.
8. 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.
9. 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.
10. J. Kienzle, W. A. Abed, F. Fleurey, J.-M. Jézéquel, and J. Klein. Aspect-oriented design with reusable aspect models. *TAOSD 7*, 6210:272–320, 2010.
11. 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.
12. T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, 1989.
13. J. Sánchez, 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.
14. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
15. Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.
16. 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.