Technical Report

A Diagrammatic Approach to Model Completion

Fazle Rabbi, Yngve Lamo, Ingrid Chieh Yu, Lars Michael Kristensen

Department of Computer Engineering
Bergen University College

September 17, 2015

Addr: Inndalsveien 28, Postbox 7030, 5020 Bergen, Norway

A Diagrammatic Approach to Model Completion

Fazle Rabbi[a,b], Yngve Lamo[a], Ingrid Chieh Yu[b], Lars Michael Kristensen[a]

[a]Bergen University College, Bergen, Norway
[b]University of Oslo, Oslo, Norway
{Fazle.Rabbi@hib.no,Yngve.Lamo@hib.no
ingridcy@ifi.uio.no,Lars.Michael.Kristensen@hib.no}

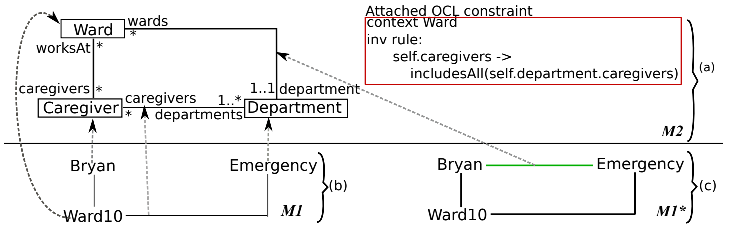
Abstract. Metamodelling plays an important role in model-driven engineering as it can be used to define domain-specific modelling languages. During the modelling phase, software designers encode domain knowledge into models which may include both structural and behavioural aspects of a system. In this paper we propose a diagrammatic approach to aid the software designer to complete partial models and thereby reduce modelling effort. The proposed technique is based on model transformations which enhance a modelling language with auto-completion. We also study the termination issue of such model transformation systems and provide sufficient condition for termination by generalizing existing work on termination of model transformation systems.

Keywords: Partial model completion, Model transformation, Diagrammatic rewriting, Graph transformation

1 Introduction

Model Driven Engineering (MDE) [13] is considered to be an efficient way of improving the quality of software and enhance software development productivity by automating repetitive, error-prone, and time-consuming tasks. In MDE, models are first–class artefacts of the software development process where models are incrementally refined starting from requirements and eventually used for code generation. Metamodels serve as the underlying foundation defining the modelling languages used to capture domain-specific knowledge and concepts.

In the process of development, software designers are often confronted with a variety of inconsistencies and/or incompleteness in the models under construction [9]. In particular, the modeller will most of the time be working with a *partial model* not conforming (i.e., not being typed by and satisfying modelling constraints) to the metamodel that defines the modelling language being used [15]. As an example, consider the following example from the healthcare domain. Figure 1(a) shows a metamodel $M2$ (in this case a class diagram) and Figure 1(b) shows a partial model (in this case an object diagram) $M1$ typed by $M2$. The typing of the nodes are represented by dotted arcs. In this example, Bryan works at Ward10; Ward10 is controlled by the Emergency department. The model $M1$ is a partial model as it is not satisfying the following domain



```
context Ward
inv rule:
    self.caregivers ->
        includesAll(self.department.caregivers)
```

Fig. 1. (a) Model $M2$, (b) a partial model $M1$ (not conforming to $M2$), (c) a completed model $M1^*$ (conforming to $M2$)

constraint: "An employee who is involved in a ward must work in the controlling department". This constraint is defined in the metamodel $M2$ by the OCL constraint shown to the upper right at Figure 1. Since Ward10 is controlled by the Emergency department, Bryan must work at the Emergency department.

In the above example, the partial model $M1$ can be transformed into a model conforming to the metamodel $M2$ by adding an arc from the Emergency department to Bryan which is missing in $M1$. Figure 1(c) shows a model $M1^*$ that conforms to the metamodel $M2$. Clearly, the productivity of the modeller could be improved by providing editing support that could either automatically add such missing model elements or make suggestions based on *completion rules* to assist the modeller in completing the model. In many respects, this idea is similar to code completion features as found in IDEs. More generally, modelling effort could be reduced by providing editing support for automated rewriting of models so that they conform to the modelling language used. Such rewriting may also involve the deletion of model elements.

The contribution of this paper is a framework for rewriting partial (incomplete) models so that they conform to the underlying metamodel. Our approach is based on rewriting of models by means of model transformation rules which supports both addition and deletion of model elements. We refer to them as *completion rules*. The proposed framework has been developed as an extension of the Diagram Predicate Framework (DPF) [7] which supports multilevel metamodelling. This approach is, however, general and could be used for other modelling frameworks as well. DPF is a language independent formalism for defining metamodelling hierarchies which provides an abstract visualization of concrete constraints. In the extended DPF, one can graphically specify completion rules. Our framework exploits the locality of model transformation rules and provides a foundation that enables automated tool-support to increase modelling productivity. In order to guarantee termination of the rewriting, we provide a set of sufficient termination criteria. A link to the prototype implementation of the proposed framework is available at *http://dpf.hib.no* where one can graphically design modelling artefacts in a web browser, apply the completion rules and check for termination.

The rest of this paper is organised as follows. Section 2 provides background knowledge on DPF, section 3 provides an outline of the partial model completion with an example, section 4 formalizes the concept of diagrammatic rewriting systems, and section 5 presents sufficient termination criteria. In section 6, we briefly discuss some related work. We assume that the reader is familiar with basic category theory and graph transformation systems[2], [6].

2 Diagrammatic Modelling with DPF

In order to work with partial models and completion rules in a formal framework; we extend the Diagram Predicate Framework (DPF) [10]. DPF provides a formal diagrammatic approach to metamodelling based on category theory and model transformations. In the DPF approach, models at any level are formalised as diagrammatic specifications which consist of graphs and diagrammatic constraints. Figure 2 shows an example of a DPF model (specification) $\mathfrak{S}_1$ with its metamodel (specification) $\mathfrak{S}_0$. The graphs represent the structure of the models; constraints are added into the structure by predicates. The metamodel (specification) $\mathfrak{S}_0$ provides the typing for the model (specification) $\mathfrak{S}_1$. Nodes in model (specification) $\mathfrak{S}_1$ are either of type Class or DataType; edges in $\mathfrak{S}_1$ are either of type Reference or Attribute. Table 1 shows a list of predicates used for constraining the specification $\mathfrak{S}_1$. Each predicate has a name (p), a shape graph ($\alpha(p)$), a visualisation and a semantic interpretation. The semantic of a predicate is provided by a set of instances. The (atomic) constraining constructs which are available for the users of the modelling language are provided in the signature Σ_i. A signature consists of a collection of diagrammatic predicates. Table 2 shows how the predicates are constraining the specification $\mathfrak{S}_1$ by a graph homomorphism $\delta : \alpha(p) \to S$ from the shape graph to the specification.

In DPF, a modelling language is formalised as a modelling formalism $(\Sigma_i, \mathfrak{S}_i, \Sigma_{i-1})$ where i and $i-1$ represent two adjacent modelling levels. The corresponding metamodel of the modelling language is represented by the specification $\mathfrak{S}_i$ which has its constraints formulated by predicates from the signature Σ_{i-1}. A DPF metamodelling hierarchy consists of (a possible stack of) metamodels, models, and instances of models. A metamodel specification determines a modelling language, the specification of a model represents a software system, and the instances of models typically represent possible states of a software system. In this paper, we present a new type of DPF construct named *completion rules* based on model transformation rules. The completion rules are connected to (domain) constraints.

3 Model Completion Example

The completion rules defined in a metamodel are used to complete a partial model. We introduce requirement R1-R5 for an envisioned system from the healthcare domain. These requirements specify the constraints of a system. Completion rules are used to automatically complete a partial model not satisfying these constraints.

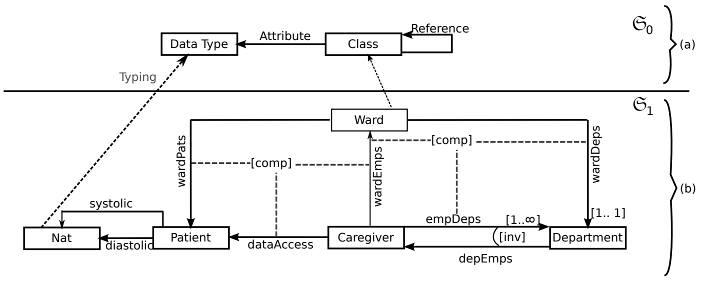


Fig. 2. a) Metamodel $\mathfrak{S}_0$, b) Model $\mathfrak{S}_1$

Table 1. Predicates of a sample signature Σ_0

p	$\alpha^{\Sigma_0}(p)$	visualisation	Semantic Interpretation
[mult(n,m)]	$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \end{tikzcd}$	$\begin{tikzcd} \boxed{X} \arrow[r, "f", "{[n..m]}"'] & \boxed{Y} \end{tikzcd}$	f must have at least n and at most m instances. Formally, $\forall x \in X : m \leq \lvert f(x) \rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
[inverse]	$\begin{tikzcd} 1 \arrow[r, bend left, "f"] & 2 \arrow[l, bend left, "g"] \end{tikzcd}$	$\begin{tikzcd} \boxed{X} \arrow[r, bend left, "f"] \arrow[r, phantom, "{[inv]}"] & \boxed{Y} \arrow[l, bend left, "g"] \end{tikzcd}$	For each instance of f there exists an instance of g or vice versa. Formally, $\forall x \in X, \forall y \in Y : y \in f(x)$ iff $x \in g(y)$
[composite]	$\begin{tikzcd} 2 \arrow[dr, "f"] & \\ 1 \arrow[u, "g"] \arrow[r, "h"'] & 3 \end{tikzcd}$	$\begin{tikzcd} \boxed{Y} \arrow[dr, "f"] & \\ \boxed{X} \arrow[u, "g"] \arrow[r, "h"'] \arrow[r, phantom, "{[comp]}", shift left=3] & \boxed{Z} \end{tikzcd}$	For each instance of $(g; f)$, there exists an instance of h. Formally, $\forall x \in X : \bigcup \{f(y) \mid y \in g(x)\} \subseteq h(x)$

Table 2. The set of atomic constraints $C^{\mathfrak{S}_1}$ of $\mathfrak{S}_1$

(p, δ)	$\alpha^{\Sigma_0}(p)$	$\delta(\alpha^{\Sigma_0}(p))$
$([mult(1, \infty)], \delta_1)$	$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \end{tikzcd}$	$\begin{tikzcd} \boxed{\text{Caregiver}_1} \arrow[r, "\text{empDeps}_f"] & \boxed{\text{Department}_2} \end{tikzcd}$
$([mult(1, 1)], \delta_2)$	$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \end{tikzcd}$	$\begin{tikzcd} \boxed{\text{Ward}_1} \arrow[r, "\text{wardDeps}_f"] & \boxed{\text{Department}_2} \end{tikzcd}$
$([inverse], \delta_4)$	$\begin{tikzcd} 1 \arrow[r, bend left, "f"] & 2 \arrow[l, bend left, "g"] \end{tikzcd}$	$\begin{tikzcd} \boxed{\text{Caregiver}_1} \arrow[r, bend left, "\text{depEmps}_f"] & \boxed{\text{Department}_2} \arrow[l, bend left, "\text{empDeps}_g"] \end{tikzcd}$
$([composite], \delta_5)$	$\begin{tikzcd} 2 \arrow[dr, "f"] & \\ 1 \arrow[u, "g"] \arrow[r, "h"'] & 3 \end{tikzcd}$	$\begin{tikzcd} \boxed{\text{Ward}_2} \arrow[dr, "\text{wardDeps}_f"] & \\ \boxed{\text{Caregiver}_1} \arrow[u, "\text{wardEmps}_g"] \arrow[r, "\text{empDeps}_h"] & \boxed{\text{Department}_3} \end{tikzcd}$
$([composite], \delta_6)$	$\begin{tikzcd} 2 \arrow[dr, "f"] & \\ 1 \arrow[u, "g"] \arrow[r, "h"'] & 3 \end{tikzcd}$	$\begin{tikzcd} \boxed{\text{Ward}_2} \arrow[dr, "\text{wardPats}_f"] & \\ \boxed{\text{Caregiver}_1} \arrow[u, "\text{wardEmps}_g"] \arrow[r, "\text{dataAccess}_h"] & \boxed{\text{Patient}_3} \end{tikzcd}$

R1. *A Caregiver (e.g., nurse, doctor) must work for at least one Department.*
R2. *A Caregiver may work for more than one Department.*
R3. *A Ward must be controlled by exactly one Department.*

R4. *A Caregiver who is involved in a Ward, must work for its controlling Department.*
R5. *All Caregivers assigned to a Ward have access to the patients information who are admitted to the same Ward.*

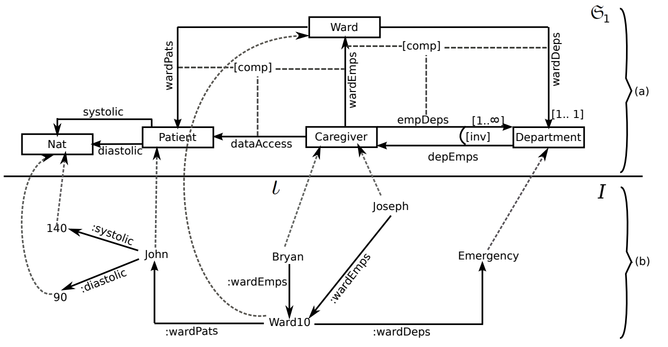


Fig. 3. a) A DPF model $\mathfrak{S}_1 = (S, C^{\mathfrak{S}_1} : \Sigma_2)$, and b) an instance (I, ι) of $\mathfrak{S}_1$

Figure 3(a) shows a DPF model (specification) $\mathfrak{S}_1$ developed from the above requirements. The metamodel (specification) of $\mathfrak{S}_1$ was shown in Figure 2(a). In $\mathfrak{S}_1$ we used the signature Σ_0 from Table 1 to define the set $C^{\mathfrak{S}_1}$ of atomic constraints. R1 and R2 is encoded in $\mathfrak{S}_1$ by the $[mult(1,\infty)]$ predicate on the edge *empDeps*; R3 by the $[mult(1,1)]$ predicate on the edge *wardDeps*; R4 by the [*composite*] predicate on edges *wardEmps*, *wardDeps* and *empDeps* where (*wardEmps*; *wardDeps*) ⊆ *empDeps*; and R5 by the [*composite*] predicate on edges *wardEmps*, *wardPats* and *dataAccess* where (*wardEmps*; *wardPats*) ⊆ *dataAccess*.

Figure 3(b) shows a candidate instance $\iota : I \to S$ of $\mathfrak{S}_1$ (also represented as (I, ι)) where S is the underlying graph of specification $\mathfrak{S}_1$. Even though I is typed by S, (I, ι) is not a valid instance of $\mathfrak{S}_1$ since it does not satisfy the predicates $[mult(1,\infty)]$ and [*composite*]. This can be checked from the pullback operation shown in Figure 4. By performing a pullback of $\alpha^{\Sigma_0}([composite]) \xrightarrow{\delta_1} S \xleftarrow{\iota} I$, we extract the fragment of the graph that is affected by the [*composite*] constraint. The graph homomorphism ι^* (see Figure 4) is not a valid instance of the [*composite*] as the semantics of the [*composite*] predicate is not satisfied. In this example, Bryan works at Ward10; Ward10 is controlled by the Emergency department. Since Ward10 is controlled by the Emergency department, Bryan must work at the Emergency department. Also, the caregivers are supposed to work for at least one department which is missing in $\iota : I \to S$ of $\mathfrak{S}_1$. Figure 5 shows a screenshot of the editor that highlights which part of the model instance violates the constraints.

In the next section, we extend DPF with model transformation rules which allows us to incorporate model completion rules. We augment the above mentioned example with completion rules at the meta-model level that can be used

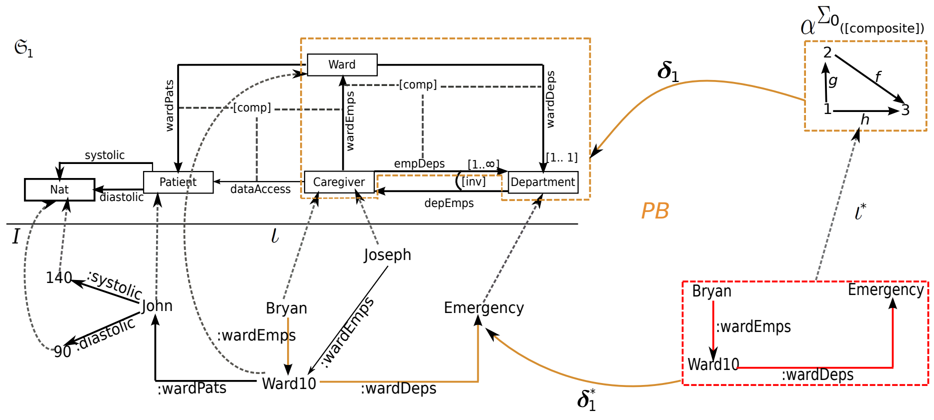


Fig. 4. Pullback $\alpha^{\Sigma_0}([composite]) \xleftarrow{\iota^*} O^* \xrightarrow{\delta_1^*} I$ of $\alpha^{\Sigma_0}([composite]) \xrightarrow{\delta_1} S \xleftarrow{\iota} I$

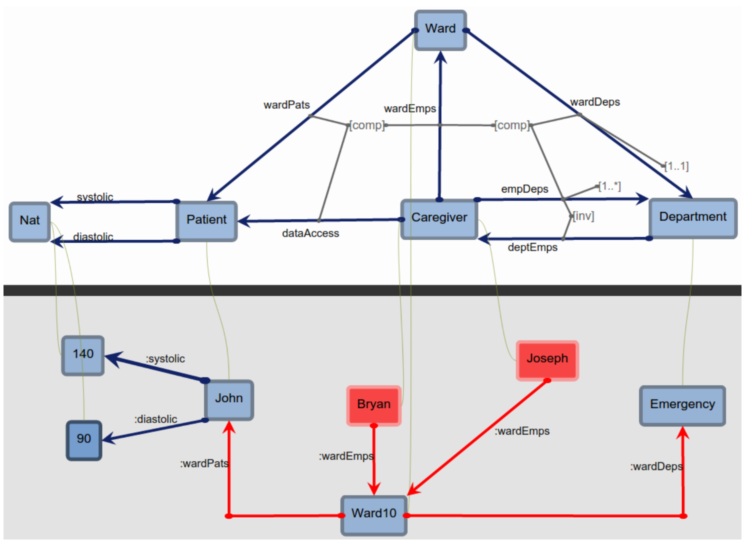
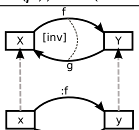
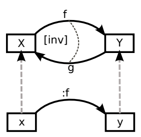
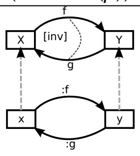
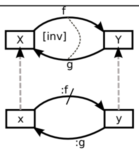
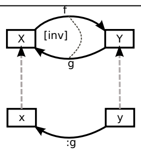
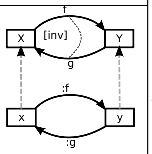
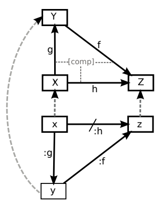
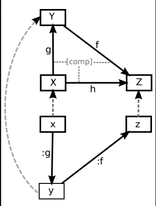
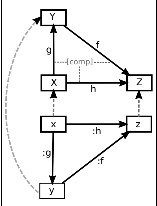


Fig. 5. An inconsistent instance (I, ι) (bottom) of $\mathfrak{S}_1$ (top)

as a basis for automatically removing the inconsistencies of the partial model (I, ι). Table 3 shows some completion rules linked to predicates. Completion rules for a predicate ensures that the predicate becomes satisfied. The semantics of completion rules are defined by coupled transformation rules [14] [3] with negative application conditions as in Table 3. Negative application conditions are typically used in graph transformations to prohibit an infinite number of rule applications. We use a special type of coupled transformations where the metamodels remain unchanged [11]. Notice that the matching patterns and negative application conditions are represented in the same diagram in Table 3 to make it more readable. Negative application conditions on model elements are represented by a strike through the corresponding model elements in the diagram to represent that they must not exist while matching.

Table 3. Completion scheme for predicates and completion rules of $\mathfrak{S}_1$

C_p	$\zeta(C_p)$			Interpretation
	$(N \rightarrow \alpha^{\Sigma}(p)) \leftarrow (L \rightarrow \alpha^{\Sigma}(p))$	$(K \rightarrow \alpha^{\Sigma}(p))$	$(R \rightarrow \alpha^{\Sigma}(p))$	
[inv-com$\rangle_{[inv]}$				derive an edge $y \xrightarrow{:g} x$ (if it does not exist) from the existence of an edge $x \xrightarrow{:f} y$ or vice versa.
				
[comp-com$\rangle_{[comp]}$				derive an edge $x \xrightarrow{:h} z$ (if it does not exist) from the existence of edges $x \xrightarrow{:g} y$ and $y \xrightarrow{:f} z$.

By applying the completion rules from Table 3 on (I, ι), the partial model is updated to become a model, (I^{*}, ι) which conforms to its metamodel. Once we apply the completion rules for $[inv - com\rangle_{[inv]}$ and $[comp - com\rangle_{[comp]}$ on (I, ι), we obtain the instance of specification $\mathfrak{S}_1$ shown in Figure 6. Bold arcs (also shown with a green color) in Figure 6 represent the additional edges derived from the application of completion rules. We do not have deleting rules in this example, but in general it is possible to have deletion rules to repair e.g., multiplicity constraints [16].

4 Diagrammatic Model Completion

Our diagrammatic rewriting system is based on completion rules. Completion rules are typed coupled transformation rules where type graphs are not changed by the transformation. The rules are linked to predicates and they are applied to a partial model to correct inconsistencies. We use the standard double-pushout (DPO) approach [6] for defining completion rules. In this section we give a formal definition of completion rules and formalize its application and matching.

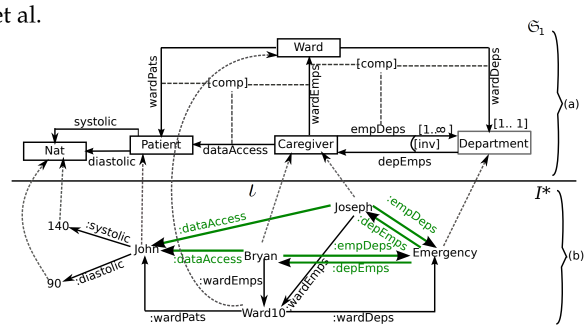


Fig. 6. a) A specification $\mathfrak{S}_1 = (S, C^{\mathfrak{S}_1} : \Sigma_0)$ and b) an instance (I^*, ι) of $\mathfrak{S}_1$

Definition 1 (Completion scheme of a predicate and completion rules) *Let $\Sigma = (P^\Sigma, \alpha^\Sigma)$ be a signature. A completion scheme C_p for a predicate $p \in P^\Sigma$, is given by a symbol cp, and a set of completion rules $\zeta(C_p)$ where the meta-models remain unchanged. A rule $r \in \zeta(C_p)$ of a completion scheme of a predicate C_p has a matching pattern (L), a gluing condition (K), a replacement pattern (R), and an optional negative application condition, NAC ($n : L \longrightarrow N$) where L, N, K, R are all typed by the arity $\alpha^\Sigma(p)$ of the predicate p.*

A completion rule as defined in Definition 1 is illustrated in the following figure. The matching pattern and replacement pattern are also known as left-hand side and right-hand side of a rule, respectively. In the figure, k and l are injective morphisms. A completion rule is nondeleting if $K = L$. In this case we have an injective morphism $L \hookrightarrow R$.

$$\begin{tikzcd}
\alpha^\Sigma(p) & \alpha^\Sigma(p) \arrow[l, "id"'] & \alpha^\Sigma(p) \arrow[l, "id"'] \arrow[r, "id"] & \alpha^\Sigma(p) \\
N \arrow[u, dotted, "\iota_N"] & L \arrow[l, "n"] \arrow[u, dotted, "\iota_L"'] & K \arrow[l, "k"] \arrow[r, "l"'] \arrow[u, dotted, "\iota_K"'] & R \arrow[u, dotted, "\iota_R"']
\end{tikzcd}$$

Figure 7 illustrates a rule associated with the [*composite*] predicate where the graphs L, N, K, R are all typed by the arity of the [*composite*] predicate.

Since the rules are not directly linked to a model, they can be reused by constraining a model with appropriate predicates.

Definition 2 (Match of a completion rule). *Let C_p be a completion scheme of a predicate with a set of completion rules $\zeta(C_p)$. A match (δ, m) of a rule $r \in \zeta(C_p)$ is given by an atomic constraint $\delta : \alpha^\Sigma(p) \longrightarrow S_{i-1}$ and a match $m : L \longrightarrow S_i$ such that constraint δ and match m together with typing morphisms $\iota_L : L \longrightarrow \alpha^\Sigma(p)$ and $\iota_{S_i} : S_i \longrightarrow S_{i-1}$ constitute a commuting square: $\iota_L; \delta = m; \iota_{S_i}$ as in the diagram below.*

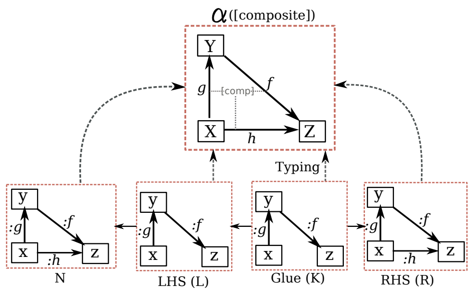


Fig. 7. A transformation rule linked to the [*composite*] predicate

$$\begin{tikzcd}
\alpha^{\Sigma}(p) \arrow[r, "\delta"] & S_{i-1} \\
L \arrow[u, dotted, "\iota_L"] \arrow[r, "m"] \arrow[ur, phantom, "="] & S_i \arrow[u, dotted, "\iota_{S_i}"']
\end{tikzcd}$$

Now we consider the satisfaction of a negative application condition of a completion rule. A rule is applied as long as it satisfies its negative application condition which is typically expressed by a graph structure in a graph transformation system.

Definition 3 (Satisfaction of a NAC of a completion rule) *A match (δ, m) of a rule $r \in \zeta(C_p)$ satisfies a NAC $(n : L \rightarrow N)$ if there does not exist an injective graph morphism $q : N \rightarrow S_i$ with $n; q = m$ such that the typing morphisms $\iota_N : N \rightarrow \alpha^{\Sigma}(p)$ and $\iota_{S_i} : S_i \rightarrow S_{i-1}$ constitute a commuting square: $\iota_N; \delta = q; \iota_{S_i}$ as shown in the diagram below. This is written $(\delta, m) \models NAC$.*

$$\begin{tikzcd}
\alpha^{\Sigma}(p) \arrow[rr, bend left, "\delta"] & \alpha^{\Sigma}(p) \arrow[l, "id"] \arrow[r, "\delta"'] & S_{i-1} \\
N \arrow[u, dotted, "\iota_N"] \arrow[rr, bend right, "q"', "\neq"] & L \arrow[l, "n"'] \arrow[u, dotted, "\iota_L"'] \arrow[r, "m"] & S_i \arrow[u, dotted, "\iota_{S_i}"']
\end{tikzcd}$$

Definition 4 (Application of a completion rule) *Let C_p be a completion scheme of a predicate with a set of completion rules $\zeta(C_p)$. A rule $r = ((N \rightarrow \alpha^{\Sigma}(p)) \leftarrow (L \rightarrow \alpha^{\Sigma}(p)) \leftarrow (K \rightarrow \alpha^{\Sigma}(p)) \rightarrow (R \rightarrow \alpha^{\Sigma}(p))) \in \zeta(C_p)$ transforms a partial model (S_i, ι_{S_i}) to $(S_i^*, \iota_{S_i^*})$ if there exists a match (δ, m) where $(\delta, m) \models NAC$. The transformation consist of two commuting cubes and two pushout diagrams (PO1 and PO2) as shown below:*

$$\begin{tikzcd}
\alpha^{\Sigma}(p) \arrow[dr, "\delta"] & & \alpha^{\Sigma}(p) \arrow[ll, "id"'] \arrow[dl, "\delta"'] & & \alpha^{\Sigma}(p) \arrow[ll, "id"'] \arrow[rr, "id"] \arrow[dl, "\delta"'] & & \alpha^{\Sigma}(p) \arrow[dl, "\delta"'] \\
 & S_{i-1} & & S_{i-1} \arrow[ll, "id'"'] \arrow[rr, "id'"] & & S_{i-1} & \\
N \arrow[uu, dotted, "\iota_N"] \arrow[dr, "q"'] & & L \arrow[ll, "n"] \arrow[uu, dotted] \arrow[dl, "m"] \arrow[dl, phantom, "\neq"'] \arrow[dr, phantom, "(PO1)"] & & K \arrow[ll] \arrow[rr] \arrow[uu, dotted] \arrow[dl] \arrow[dr, phantom, "(PO2)"] & & R \arrow[uu, dotted] \arrow[dl, "m^*"] \\
 & S_i \arrow[uu, dotted, "\iota_{S_i}"] & & S_i' \arrow[ll] \arrow[rr] \arrow[uu, dotted] & & S_i^* \arrow[uu, dotted, "\iota_{S_i^*}"'] &
\end{tikzcd}$$

Note that $(S_i^*, \iota_{S_i^*})$ could be a partial model and in that situation, further application of completion rules are required to obtain a a model that satisfies all the constraints. Our framework supports the execution of rules in two different ways: one can interactively apply the rules where the user guides the execution order; or automatically considering all possible orderings of rule execution. In the latter case, it is important to perform an analysis to ensure that the execution of the transformation rules will eventually terminate.

5 Termination Criterion and Layered Completion Rules

The formulation of the completion rules allows negative application conditions which makes rules very expressive. One issue with such expressive rules is that the rewriting systems may suffer from non-termination. In [6], Ehrig et al. presented a termination criterion for typed graph transformation system. The authors proposed layered typed graph grammar (layered GG) which always terminates. In a layered typed graph grammar, transformation rules are distributed across different layers. The transformation rules of a layer are applied as long as possible before going to the next layer. They distinguished between deletion and nondeletion layers where all transformation rules in deletion layers delete at least one element and all transformation rules in nondeletion layers do not delete any elements, but the rules have negative application conditions. The theory provided in [6] on the termination of layered typed graph grammar is based on a type set which is given by a type graph. The specified layer conditions (see [6]) that need to be satisfied by each production layer k. A production layer (pl) maps each rule to a layer number. Similarly a creation layer (cl) and deletion layer (dl) maps each element from the type graph to a layer number. The deletion layer conditions require that for a deletion layer k and any rule r with production layer $pl(r) = k$, the following is satisfied:

- r must delete at least one element
- r cannot delete an element x if x has been created in layer k

The non-deletion layer conditions ensures that, for a non-deletion layer k' and any rule r' with production layer $pl(r') = k'$, the following is satisfied:

- r' cannot delete elements

– r' cannot be applied twice with the same match
– r' cannot use an element x for the match if x has been created in layer k'

The layer conditions are specified in terms of the type of the elements being created and/or deleted by the transformation rules. The layer conditions imply that the creation layer of an element of type t must precede its deletion layer. Since the layered conditions provide only sufficient criterion for termination, for some terminating graph transformation systems a layered graph grammar does not exist. For example, consider the graph transformation system shown in Figure 8. This example shows a graph transformation system with a type graph TG, two transformation rules ($p1$, $p2$), and an initial graph G_0. Transformation rule $p2$ performs the transitive closure for a *ancestor* relation.

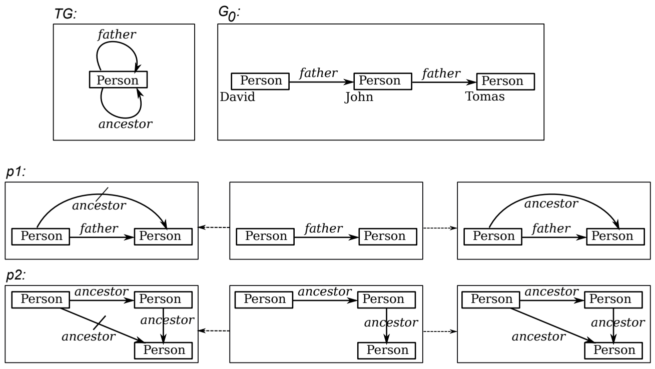


Fig. 8. An example of graph transformation system with nondeleting rules

For this example, a set of layers does not exist such that all the conditions are fulfilled. Figure 9 shows the derivation graph of the graph transformation system of Figure 8. $p1$ and $p2$ are nondeleting rules and therefore have to belong to nondeletion layer(s). Here $\forall p \in \{p1, p2\}, pl(p)$ is a production layer with $0 \leq pl(p) \leq k_0(pl(p), k_0 \in \mathbb{N})$, where $k_0 + 1$ is the number of layers; and for type *ancenstor*, there is a creation layer, $cl(ancenstor) \in \mathbb{N}$. The nondeletion layer conditions imply the following inequalities which cannot be fulfilled:
$pl(p1) < cl(ancestor) \leq pl(p2) < cl(ancestor)$.

In Figure 10 we provide another example of a graph transformation system with deleting rules that cannot be layered. In this example we have two productions $p1$, and $p2$. $p1$ specifies that if a patient specifies privacy restriction to a caregiver, then the caregiver cannot access the patients health record. $p2$ specifies that any caregiver who works in the emergency department can access a patients health record who are in acute pain. Figure 10 shows an initial graph G_0 where the patient *John* has a privacy policy that restricts caregiver *Joseph* from viewing *John*'s health record. Since *John* is in *AcutePain*, *Joseph* must see *John*'s health record in order to deal with the emergency situation. Thus apply-

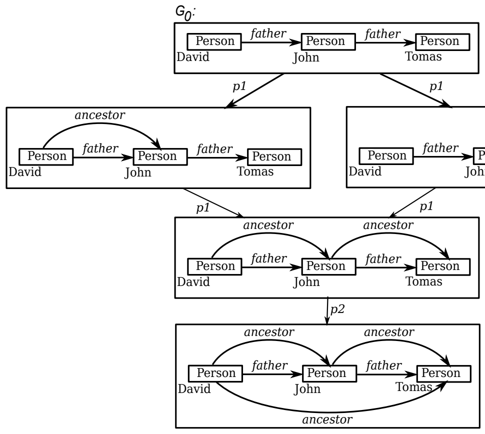


Fig. 9. Derivation graph

ing $p2$ after $p1$ gives *Joseph* access to the patients health record and after that the derivation process terminates. Here $\forall p \in \{p1, p2\}, pl(p)$ is a production layer; and for type *dataAccess*, there is a creation layer $cl(dataAccess) \in \mathbb{N}$, and a deletion layer $dt(dataAccess) \in \mathbb{N}$. The deletion layer conditions imply the following inequalities which cannot be fulfilled:

$dl(dataAccess) \leq (p2) < cl(dataAccess)$.

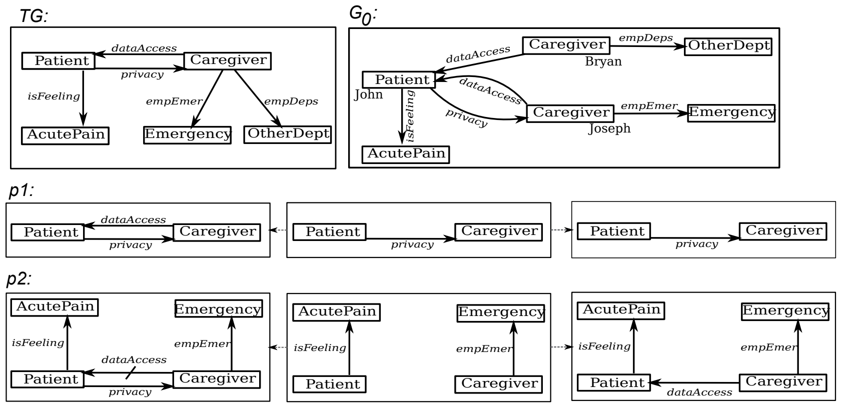


Fig. 10. An example of graph transformation system with deleting rules

Below we define DPF specification with layered completion rules that overcomes this limitations and shows termination of a more general set of graph transformation systems. As part of this definition, we generalize the termination criterion of [6].

Definition 5 (DPF specification with layered Completion rules) *A DPF specification $\mathfrak{S}_i = (S_i, C^{\mathfrak{S}_i} : \Sigma_{i-1})$ with a set of completion rules $R^{\mathfrak{S}_i}$ is layered if:*

1. *$R^{\mathfrak{S}_i}$ is layered, i.e. for each $r \in R^{\mathfrak{S}_i}$ there is a production layer $pl(r)$ with $0 \le pl(r) \le k_m$ $(pl(r), k_m \in \mathbb{N})$, where $k_m + 1$ is the number of layers in $\mathfrak{S}_i$.*
2. *The graph S_i and the set of completion rules $R^{\mathfrak{S}_i}$ are finite.*
3. *For each element $x \in S$, where S is derivable from S_i via a transformation, there is a creation layer $cl(x) \in \mathbb{N}$, and each layer k satisfies the following conditions for all $r \in R_k^{\mathfrak{S}_i}$:*

(a) r either deletes an element x of type t or r has NAC $(n : L \to N)$ where $N \setminus L \neq \emptyset$ and there is an injective morphism $n' : N \hookrightarrow R$
(b) r uses an element x for the match: $x \in m(L) \implies cl(x) \le k$
(c) r creates a node $x \implies cl(x) = k + 1$
(d) r creates an edge $e \implies cl(e) = k$
(e) $\exists \neg R_k \subseteq R_k^{\mathfrak{S}_i}$ such that rules from R_k can be applied indefinitely in a loop

For each element x in the initial graph $G_0 = S_i$, $cl(x) = 0$. By the layer conditions, only elements x with $cl(x) \le k$ can take part in matching with the left-hand side L. Therefore a newly created element x with $cl(x) = (k+1)$ cannot take part in the matching with the left hand side at layer k. However they take part in matching with N. As an example, suppose H_i is derived by the application of rule r over G_0. Further application of a rule r with $pl(r) = k$ transforms a DPF instance (H_i, ι_{H_i}) to an instance $(H_{i+1}, \iota_{H_{i+1}})$ if there exists a match (δ, m) where $m : L \hookrightarrow H_i$ with $m(L) \subset H_i$ and there does not exists an injective graph morphism $q : N \hookrightarrow H_i$ with $n; q = m$ such that $\iota_N; \delta = q; \iota_{H_i}$. Here $\forall x \in m(L) : cl(x) \le k$. The transformation is illustrated below:

$$\begin{tikzcd}
\alpha^{\Sigma}(p) \arrow[dr, "\delta"'] & & \alpha^{\Sigma}(p) \arrow[ll, "id"'] \arrow[dl, "\delta"] & & \alpha^{\Sigma}(p) \arrow[ll, "id"'] \arrow[rr, "id"] \arrow[dl, "\delta"] & & \alpha^{\Sigma}(p) \arrow[dl, "\delta"] \\
& S_{i-1} & & S_{i-1} \arrow[ll, "id'"'] \arrow[rr, "id'"] & & S_{i-1} & \\
N \arrow[uu, dotted, "\iota_N"] \arrow[dr, "q"', "\neq"] & & L \arrow[ll, "n"'] \arrow[uu, dotted] \arrow[dl, "m"] \arrow[dr, phantom, "(PO1)"] & & K \arrow[ll] \arrow[rr] \arrow[uu, dotted] \arrow[dl] \arrow[dr, phantom, "(PO2)"] & & R \arrow[uu, dotted] \arrow[dl, "m^*"] \\
& H_i \arrow[uu, dotted, "\iota_{H_i}"] & & H_i' \arrow[ll] \arrow[rr] \arrow[uu, dotted] & & H_{i+1} \arrow[uu, dotted, "\iota_{H_{i+1}}"] &
\end{tikzcd}$$

Since a newly created edge e takes part in the matching with the left hand side of a rule, this allows us to perform transitive closure operation. By the layer conditions, a set of rules that cannot be applied indefinitely in a loop are allowed in a layer. A necessary condition for looping will be presented later in this paper. A static analysis technique is also provided for the detection of loops from a set of rules.

5.1 Termination of DPF Specification with Layered Completion Rules

We start by considering necessary conditions for looping. Let G_0 be an initial finite graph and R_k be a finite set of rules of layer k. Rules in R_k can be applied indefinitely in a loop if they satisfy one of the following conditions:

- **LC1:** $r_i \in R_k$ deletes an element x of type $t \implies \exists r_j \in R_k$ such that r_j creates an element x' of type t with $cl(x') = k$.
- **LC2:** $r_i \in R_k$ has a NAC that forbids the existence of an element x of type $t \implies \exists r_j \in R_k$ such that r_j deletes an element x' of type t.

Lemma 1. *(**LC1** $\vee$ **LC2**) is a necessary condition for looping of a set of rules at layer k*

Proof: Let $G_0 = S_i$ be an initial graph typed by S_{i-1} where S_i, S_{i-1} are finite graphs. Let R_k be a finite set of rules that can be applied indefinitely in a loop at layer k. We have to show that (**LC1** $\vee$ **LC2**) must be satisfied by the rules in R_k.
According to the conditions for layer k, any rule $r \in R_k$, r either

i) deletes an element x of type t and/or
ii) r has an element x' in its $(N \setminus L)$.

Consider the first case i): the rule r has finite number of injective matches $c_r = \{(\delta, m) \mid (\delta, m) \textit{ is a match for } G_0 \rhd S_{i-1} \textit{ and } (\delta, m) \models NAC\}$. In order to apply r indefinitely in a loop during the derivation process, new elements x of type t with $cl(x) = k$ must be created. Therefore **LC1** is a necessary condition for looping in this situation.
Consider the second case ii): the rule r has finite number of injective matches $c_r = \{(\delta, m) \mid (\delta, m) \textit{ is a match for } G_0 \rhd S_{i-1} \textit{ and } (\delta, m) \models NAC\}$. For each injective match of $L \to G_0$, application of r creates an element x of type t as $N \hookrightarrow R$. Therefore, the application of rule r decreases the number of matches. In order to apply r indefinitely in a loop during the derivation process of layer k, elements x' of type t must be deleted. Therefore **LC2** is a necessary condition for looping in this situation.
The following corollary follows from Lemma 1.

Corollary 1. *A sufficient condition for loop-free rules in layer k is the negation of (**LC1** $\vee$ **LC2**)*

Detection of loop-free rules: Note that (**LC1** $\vee$ **LC2**) is a necessary condition for looping but it is not sufficient. So instead of detecting a loop from a set of rules, we provide a static analysis technique for detecting a set of rules that can be successively applied with guaranteed termination. Loop-free rules can be checked from a set of rules by constructing a table categorizing the action of rules. For each type t of a given type graph, we make two columns in a table: (i) $(N \setminus L)(t)$, and (ii) R(t). For each rule r from a given set of rules, we put a symbol from (✓, ×, *no entry*, ✓×) in each cell of the table depending on the following conditions:

Rule	$(N \setminus L)$(t)	R(t)
r	✓ - r has a NAC that forbids the existence of an element x of type t *no entry* - otherwise	✓ - r creates an element x of type t with $cl(x) = k$ × - r deletes an element of type t ✓× - r creates and deletes elements x of type t with $cl(x) = k$ *no entry* - otherwise

After constructing a table from a given set of rules R_L we perform an analysis to check loop freeness. The basic idea of the analysis is to remove a rule r from the table if r cannot be part of a loop. We delete a rule r and all its entries from a table if

1. r deletes an element of type t, but there is no rule that creates an element x of type t with $cl(x) = k$.
2. r has a NAC that forbids the existence of an element x of type t, but there is no rule that deletes an element of type t.

These two steps can be easily performed by a search in the table. A row for r is deleted if

1. R(t) has a cell with × symbol in it but there are no cells under the column R(t) with ✓
2. $(N \setminus L)$(t) has a cell with ✓ symbol in it but there are no cells under the column R(t) with ×

After performing these steps, the table contains rules that satisfies (**LC1** ∨ **LC2**) and therefore R_k may contain loops. On the other hand if the table is empty, there does not exists any looping in R_k.
The following table is constructed from the example provided in Figure 10. We did not show all columns for type graph TG as the cells contains no entry for them.

Rule	$(N \setminus L)$(privacy)	R(privacy)	$(N \setminus L)$(dataAccess)	R(dataAccess)
p1				×
p2		×	✓	✓

In this table, rule $p2$ deletes an element of type *privacy* but there is no rule that creates elements of type *privacy*. Therefore all the entries for $p2$ are removed. All the entries of $p1$ are removed as well from the table as after the deletion of $p2$ there are no rules that can create elements of type *dataAccess*. Thus the table becomes empty and ensures that the rules have no looping and therefore they can be executed in the same layer.

Theorem 1 (termination of loop-free rules). *An empty table obtained by loop free rule detection analysis for a set of rules E implies that the execution of E will terminate for any finite size initial graph.*

Proof: We will prove this by contradiction. Assume that an empty table is obtained by loop free rule detection analysis. Let $G_0 = S_i$ be an initial graph typed by S_{i-1} where S_i, S_{i-1} are finite graphs. Suppose, for contradiction, that E is a finite set of rules that can be applied indefinitely in a loop at layer k; therefore accordingly Lemma 1, E must satisfy (**LC1** ∨ **LC2**).

Assume that E satisfies **LC1**: there must exist at least one rule $r_i \in E$ that deletes an element of type t and a rule $r_j \in E$ that creates an element x of type t with $cl(x) = k$. Therefore there must exist two entries in the table under column $R(t)$- one with ✓ and one with × which makes the table non-empty by loop free rule detection analysis.

Assume that *E* satisfies **LC2**: similar arguments can be presented to show that the table cannot be empty.

In either situation the table cannot be empty if *E* can be executed in a loop. Therefore an empty table implies that *E* will terminate.

6 Related and Future Work

Partial models have been used in MDE for various purposes. It has been used for modelling with uncertainty by Salay et al. in [12]. They proposed four types of partiality that can be defined in a modelling language–independent way. Their definition of metamodel consists of a First Order Logic theory that includes a set of sentences representing the well–formedness constraints. Modelling with uncertainty is essential in situations such as: i) the requirements are not clearly specified, ii) alternative resolution to inconsistency is present, and iii) stakeholders opinions differ. In this article, we focus on precise modelling of systems with metamodelling assuming the requirement is clearly specified.

An approach similar to our work was presented in [15] where the authors presented a methodology to automatically solve partial model completion problems. They transformed a partial model, its metamodel, and additional constraints to a constraint logic program (CLP). The metamodel specifies the domain of possible assignments for each and every property. This information is being used by the CLP to complete a partial model object. They provided an algorithm to generate code that assigns a domain of values to the attributes and relationships. When the CLP is queried, a solver selects values from the domain of each property such that the conjunction of all the constraints is satisfied. In our approach, we use multilevel metamodelling for specifying domain models. We use diagrammatic approach to define completion rules and use model transformations to derive a complete model instance that satisfies the predicate constraints.

An automated approach for detecting and tracking inconsistencies in real time was presented in [5]. The author listed 24 consistency rules that cover the most significant concerns of keeping sequence diagrams consistent with class and statechart diagrams and evaluated the rules on UML design models. While their approach is based on the syntactical constraints of UML diagrams, ours is based on domain specific diagrammatic constraints. In our approach, the completion rules are graphically formulated, customizable, and vary from one domain to another.

The eMoflon tool [1] is built on the Eclipse Modelling Framework (EMF), using Ecore for metamodelling that supports rule-based unidirectional and bidirectional model transformation. eMoflon is featured with MOF 2.0 compliant code generation and concentrates on bidirectional model transformations with triple graph grammars and their mapping to unidirectional transformations. In contrast, we proposed metamodelling with partial model completion based on coupled model transformation and provided termination analysis for such model transformation.

In [4], Bottoni et al. proposed an approach to the identification of a sufficient criterion for termination of graph transformations with negative application conditions. Their approach is based on labelled transition systems and they use a double pushout approach to graph transformation. Their approach is focused on the termination of a single non-deleting rule. A graph transformation system typically have multiple rules and termination becomes a complex issue with multiple rules. Ehrig et al. presented a layered graph transformation systems where rules are grouped into different layers [6]. Mixing deleting and non-deleting rules in a layer is not possible. In this paper, we generalize the layer conditions from [6] allowing that deleting and non-deleting rules remains in the same layer as long as the rules are loop-free. Furthermore, we permit a rule to use newly created edges allowing us to perform transitive closure operation.

Termination criterion for double pushout transformation that covers transitive closure was addressed in [8] by Levendovszky et al. where the authors proposed a transformation based on E-concurrent production which is an amalgamation technique for concurrent execution of rules. This approach is based on the construction of concurrent rules from different combination of productions. A disadvantage of their approach is that it is hard to find all the possible sequences of graph productions, and prove that the corresponding series of cumulative 'left-hand side' exceeds all limits which is required to show termination.

An application of partial model completion is in the evolving software engineering process. In many cases, software models need to migrate because of the evolution of software requirements and/or modelling languages [16]. The concept of partial models is applicable also in this setting if accompanied by rewriting rules and would potentially be able to automate the model migration process.

References

1. A. Anjorin, M. Lauder, S. Patzina, and A. Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. In *3. Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe2011)*, 2011.
2. M. Barr and C. Wells, editors. *Category Theory for Computing Science, 2nd Ed.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.
3. S. Becker. Coupled model transformations. In *Proceedings of the 7th International Workshop on Software and Performance*, WOSP '08, pages 103–114, NY, 2008. ACM.
4. P. Bottoni and F. Parisi-Presicce. A termination criterion for graph transformations with negative application conditions. *ECEASST*, 30, 2010.
5. A. Egyed. Automatically detecting and tracking inconsistencies in software design models. *Software Engineering, IEEE Transactions on*, 37(2):188–204, March 2011.
6. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006.
7. Y. Lamo, X. Wang, F. Mantz, W. MacCaull, and A. Rutle. Dpf workbench: A diagrammatic multi-layer domain specific (meta-)modelling environment. In R. Lee, editor, *Computer and Information Science 2012*, volume 429 of *Studies in Computational Intelligence*, pages 37–52. Springer, 2012.

8. T. Levendovszky, U. Prange, and H. Ehrig. Termination criteria for dpo transformations with injective matches. *Electr. Notes Theor. Comput. Sci.*, 175(4):87–100, 2007.
9. T. Mens and R. Van Der Straeten. Incremental resolution of model inconsistencies. In *Recent Trends in Algebraic Development Techniques*, volume 4409 of *LNCS*, pages 111–126. Springer, 2007.
10. A. Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, Norway, 2010.
11. A. Rutle, W. MacCaull, H. Wang, and Y. Lamo. A metamodelling approach to behavioural modelling. In *Proceedings of the Fourth Workshop on Behaviour Modelling - Foundations and Applications*, BM-FA '12, pages 5:1–5:10, NY, USA, 2012. ACM.
12. R. Salay, M. Famelis, and M. Chechik. Language independent refinement using partial modeling. In *Fundamental Approaches to Software Engineering*, volume 7212 of *LNCS*, pages 224–239. Springer, 2012.
13. D. C. Schmidt. Guest editor's introduction: Model-driven engineering. 39(2):25–31, Feb. 2006.
14. C. Schulz, M. Löwe, and H. König. A categorical framework for the transformation of object-oriented systems: Models and data. *J. Symb. Comput.*, 46(3):316–337, Mar. 2011.
15. S. Sen, B. Baudry, and D. Precup. Partial model completion in model driven engineering using constraint logic programming. In *INAP'07*, Germany, 2007.
16. G. Taentzer, F. Mantz, T. Arendt, and Y. Lamo. Customizable model migration schemes for meta-model evolutions with multiplicity changes. In *16th International Conference, MODELS 2013, Miami, Proceedings*, volume 8107 of *LNCS*, pages 254–270. Springer, 2013.