Composition of Multilevel Domain-Specific Modelling Languages

Alejandro Rodríguez[a,*], Fernando Macías[d], Francisco Durán[c], Adrian Rutle[a], Uwe Wolter[b]

[a] *Western Norway University of Applied Sciences, Bergen, Norway*
[b] *University of Bergen, Bergen, Norway*
[c] *ITIS Software, University of Málaga, Málaga, Spain*
[d] *IMDEA Software Institute, Madrid, Spain*

Abstract

Multilevel Modelling (MLM) approaches make it possible for designers and modellers to work with an unlimited number of abstraction levels to specify their domain-specific modelling languages (DSMLs). To fully exploit MLM techniques, we need powerful model composition operators. Indeed, the composition of DSMLs is becoming increasingly relevant to the modelling community either because some DSMLs may share commonalities that we want to make reusable, or because we want to facilitate interoperability between DSMLs. In this paper, we propose a composition mechanism for structure and behaviour of multilevel modelling hierarchies. Our approach facilitates the inclusion of additional features while keeping a clear separation of concerns that enhances modularity. We provide a formal semantics of the constructions based on category theory and graph transformations and show their use in practice on a case study.

Keywords: Model-driven software engineering, Domain-specific modelling languages, Multilevel Modelling, Composition, Category theory, Graph theory, Graph transformations

1. Introduction

Multilevel Modelling is a prominent research area where models and their specifications can be organised into several levels of abstraction [1, 2]. Although there exist several approaches for MLM (see [3, 4, 5, 6] for some of them), they all share the idea of not limiting the number of levels that designers can use to specify their modelling languages. This restriction is present in traditional Model-Driven Software Engineering (MDSE) approaches which are

*Corresponding author
Email addresses: arte@hvl.no (Alejandro Rodríguez), fernando.macias@imdea.org (Fernando Macías), duran@lcc.uma.es (Francisco Durán), Adrian.Rutle@hvl.no (Adrian Rutle), uwe.wolter@uib.no (Uwe Wolter)

based on the Object Management Group (OMG) 4-layer architecture such as the Unified Modelling Language (UML) [7] and the Eclipse Modelling Framework (EMF) [8, 9]. Like traditional MDSE approaches, MLM uses abstractions and modelling techniques to tackle the continually increasing complexity of software by considering models as first-class entities throughout the software engineering life cycle. Despite the success of MDSE approaches in terms of quality and effectiveness gains [10], modellers can only make use of two levels of abstraction to specify their systems: one for (meta)models and one for their instances. Model designers might find this limitation too restrictive. Moreover, these limitations may lead to complications like model convolution, accidental complexity and mixing concepts belonging to different domains (see, e.g., [11, 12, 13] for discussions on this).

One of the most successful applications of MDSE is in the construction of (industrial) DSMLs [9]. DSMLs are modelling languages tailored to specific areas which are meant to be easily understood and used by domain experts. Thus, such challenges become more prevalent in the case of defining DSMLs, since variations on general purpose languages (i.e., to specify different refinements oriented to the different domains) would require further specialisations on the metamodels.

The MLM community has demonstrated that MLM is a successful approach in areas such as process modelling and software architecture domains [11, 14, 15]. Furthermore, MLM techniques are excellent for the creation of DSMLs, especially when focusing on behavioural languages, since behaviour is usually defined at the metamodel level while it is executed, at least, two levels below at the instance level [16, 17].

Although DSMLs are conceived to describe and abstract different concrete domains, we may find many similarities between existing DSMLs. In fact, the research community in software language engineering has proposed the notion of *Language Product Lines Engineering* (LPLE) with the goal of constructing software product lines where the products are languages [18]. The key aspect of their approach is the definition of *language features* that encapsulate a set of language constructs representing certain DSML functionalities. Usually, one can detect that some DSMLs share certain commonalities coming from similar modelling patterns that can be abstracted and reused across several other languages. Interoperability and reusability can therefore be achieved by advocating modularisation and composition techniques.

We have observed that several DSMLs can benefit from each other by composing them, resulting into a more complete system specification. To cope with this, we present an alternative approach to handle composition based on multiple typing which we compare with the standard way of facing composition through a *merge* operator. Traditionally, frameworks had to craft, in a tedious, ad-hoc and (usually) non reusable way, their own composition operators. Further research in this direction had raised more standard and widely accepted composition mechanisms, such as the merge operator or through direct linking among modules [18, 19]. Taking advantage of MLM and inspired by the concept of *language feature*, we present in this paper mechanisms based on our MLM

approach and multiple typing to foster composition by defining the abstract syntax and the behavioural description in a modular way, i.e., by adding/removing dimensions to a selected model or a model transformation rule. We compare our construction with the merge operator and put into practice our constructs to achieve composition by applying them to a case study where we consider a multilevel DSML for processes management and a DSML that abstracts human being notions.

The rest of the paper is organised as follows. Section 2 describes our approach for Multilevel Modelling regarding structure (Section 2.1) and operational semantics (Section 2.2). Section 3 presents our composition mechanism. After motivating this mechanism in Section 3.1, we compare it to the usual merge operator and present its categorical semantics in Section 3.2. We apply in Section 4 the formal constructions presented in Section 3.2 to a case study where we demonstrate how the composition of two different languages can be successfully managed. In Section 5, we discuss related work, and finally conclude the paper and outline directions for future work in Section 6.

2. Background: Multilevel Modelling

MLM is a recognised research area with clear advantages in several scenarios [20]. It provides the flexibility needed to avoid the use of anti-patterns, e.g., the type-object pattern described in [11, 21] when fitting several layers of abstraction into one single level. This anti-pattern appears when both the concept and the metaconcept have to be defined in the same level, leading to convolution. However, there exist several challenges within the MLM community that hamper its wide-range adoption, such as a lack of recognised standards and fundamental concepts of the paradigm, that have led to a proliferation of different multilevel tools [22, 23] without a clear consensus and focus.

The MultEcore approach for MLM combines two-level and multilevel modelling approaches and takes the best from each world with the goal of bringing standards into MLM solutions [16, 24]. Its main goal is to facilitate the specification of multilevel hierarchies which are both generic and precise [25, 24]. These ideas are reflected in the MultEcore tool. The tool enables multilevel modelling in the Eclipse Modelling Framework (EMF), allowing us to reuse the existing EMF tools and plugins [26, 27]. MultEcore provides facilities to the modeller to define both the structure and the behaviour of multilevel hierarchies.

MultEcore is designed as a set of Eclipse plugins, giving access to its mature tool ecosystem (integration with EMF) and incorporating the flexibility of MLM. In the MultEcore approach [16], the abstract syntax is provided by MLM models and the behaviour by the so-called Multilevel Coupled Model Transformations (MCMTs) [16, 25]. Using the MultEcore tool, modellers can (i) define MLM models using the model graphical editor, (ii) define MCMTs using its rule editor, and (iii) execute specific models. The execution of MultEcore models rely on a transformation of these models into Maude [28] specifications [29]. To provide a formal description of our framework and the aforementioned features, we rely on graph transformations and corresponding parts of category theory.

2.1. Multilevel Modelling in MultEcore - Structure

The MultEcore multilevel modelling approach is based on a flexible typing mechanism based on graphs. We present in this section a summary of the formalisation in [30] on which we base the semantics of our composition construction in Section 3.2. In this formalisation, models are represented as graphs, since they are a natural way of abstracting concepts and the relations among them. Each model in our approach is identified by a name and represented as directed multigraph. Graphs are defined as follows.

Definition 1 (Graph)**.** *A Graph* $G = (G^N, G^A, sc^G, tg^G)$ *consists of a set of nodes* G^N*, a set of arrows* G^A *and two maps* $sc^G : G^A \to G^N$ *and* $tg^G : G^A \to G^N$ *that assign to each arrow its source and target node, respectively. These two maps must be total for the graph to be considered valid. We use the notations* $x \xrightarrow{f} y$ *or* $f : x \to y$ *to indicate that* $sc^G(f) = x$ *and* $tg^G(f) = y$*.*

Intuitively, graphs consist of nodes and arrows. A node represents a class, and an arrow represents a relation between two classes. Hence, an arrow always connects two nodes in the same graph, and any two nodes can be connected by an arbitrary number of arrows. Relations between graphs, like typing and matching, are defined by means of *graph homomorphisms*.

Definition 2 (Graph Homomorphism)**.** *A homomorphism* $\varphi : G \to H$ *between graphs is given by two maps* $\varphi^N : G^N \to H^N$ *and* $\varphi^A : G^A \to H^A$ *such that* $sc^G; \varphi^N = \varphi^A; sc^H$ *and* $tg^G; \varphi^N = \varphi^A; tg^H$*. Note that we use the symbol* $\_;\_$ *to denote composition in diagrammatic order.*

We use the terms graph and model indistinctly. Models are distributed in *multilevel modelling hierarchies*. By a multilevel modelling hierarchy we understand a tree-shaped hierarchy of models with a single root one typically depicted at the top of the hierarchy tree. Thus, hierarchies enclose a set of models which are connected via typing relations.

Figure 1 displays a simple multilevel hierarchy containing three levels of abstraction (four if we include the reserved *Ecore* model placed at the top in level 0, Figure 1(a)). Note that each graph, except the one at the top has exactly one parent graph in the hierarchy. Then, at Level 1, we branch into two paths. The models *generic-model-1* and *generic-model-2* (Figures 1(b) and 1(c), respectively) contain three nodes and one relation each. As shown in the figure, the type of a node is indicated in an ellipse at its top left side, e.g., EClass is the type of A, B, and C in model *generic-model-1*, as well as of D, E, and F in model *generic-model-2*. The type of an arrow is written near the arrow in italic font type, e.g., EReference under G in model *generic-model-1*, and under H in model *generic-model-2*. As we see below, typing relations are graph homomorphisms. However, we use these two individual typing graphical representations to express types without filling up the hierarchy graphical representations with arrows.

A hierarchy has $n+1$ abstraction levels, where n is the maximal path length in the hierarchy tree. Levels are indexed with increasing natural numbers starting from the uppermost one, with index 0. Each graph in the hierarchy is placed at

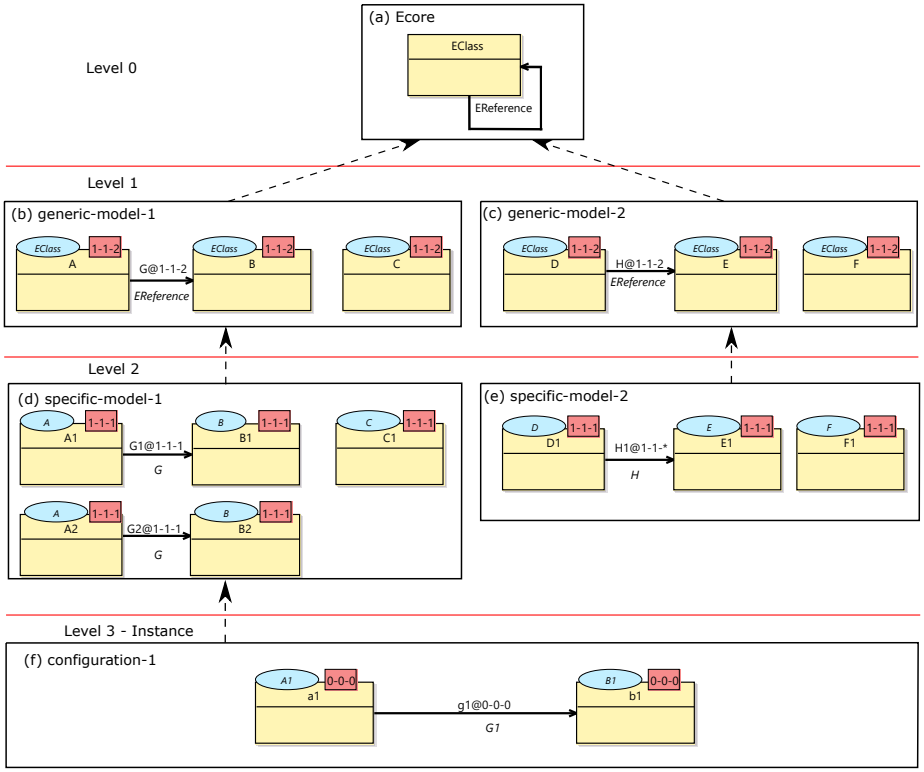


Figure 1: Multilevel hierarchy for a conceptual example

some level i, where i is the length of the path from that graph to the topmost one. To be flexible concerning abstraction levels and to support a smooth evolution of modelling descriptions, we allow certain positions in a hierarchy to be empty, i.e., filled by an empty graph. We use the notation G_i to indicate that a graph is placed at level i. For implementation reasons, we use Ecore [8] as root graph at level 0 in all example hierarchies, since Ecore is based on the concept of graph which makes it powerful enough to represent the structure of software models.

We use levels as an organisational tool, where the main rationale for locating elements in a particular level is grouping them by how abstract they are, and how reusable and useful they can be in that particular level. Thus, we encourage the *level cohesion* principle [31], that is, we recommend to organise elements that are semantically close (by means of potency and level organisation). On the contrary, we do not promote the *level segregation* principle, which establishes that level organisational semantics should be unique, i.e., aligned to one particular organisational scheme, such as *classification* or *generalisation*. We use, however, a more broad *abstraction* semantics. Furthermore, the MultEcore

tool checks correct potency and typing safeness.[1]

In Figure 1, red horizontal lines are used to indicate the separation between two consecutive levels, and upwards dashed arrows represent sequences of graphs that constitute *typing chains* G_i, G_{i-1}, ..., G_1, G_0.

For flexibility reasons, we allow typing to jump over abstraction levels, i.e., an element in graph G_i may have no type in G_{i-1} but only in one (or more) of the graphs in G_{i-2}, ..., G_1, G_0. Moreover, two different elements in the same graph may be typed by elements located in different graphs along the typing chain. To formalise this kind of flexible typing, we use *partial graph homomorphisms*.

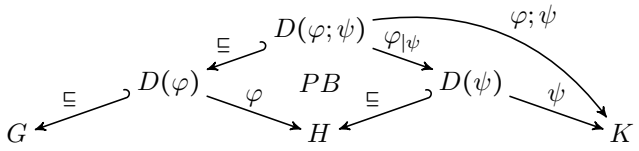
Definition 3 (Partial Graph Homomorphism)**.** *A **partial graph homomorphism** $\varphi : G \multimap H$ is given by a subgraph $D(\varphi) \sqsubseteq G$, called the **domain of definition** of φ, and a graph homomorphism $\varphi : D(\varphi) \to H$.*

To express transitivity of typing and later also compatibility of typing, we need as well the composition of partial graph homomorphisms as a partial order between partial graph homomorphisms.

Definition 4 (Composition of partial graph homomorphisms)**.** *The **composition** $\varphi;\psi : G \multimap K$ of two partial graph homomorphisms $\varphi : G \multimap H$ and $\psi : H \multimap K$ is defined as follows:*

- *$D(\varphi;\psi) := \varphi^{-1}(D(\psi))$, i.e., for all nodes $e \in G^N$ we have $e \in D(\varphi;\psi)^N$ iff $e \in D(\varphi)^N$ and $\varphi^N(e)$ in $D(\psi)^N$, and for all arrows $f \in G^A$ we have $f \in D(\varphi;\psi)^A$ iff $f \in D(\varphi)^A$ and $\varphi^A(f) \in D(\psi)^A$.*
- *$(\varphi;\psi)^N(e) := \psi^N(\varphi^N(e))$ for all $e \in D(\varphi;\psi)^N$ and $(\varphi;\psi)^A(f) := \psi^A(\varphi^A(f))$ for all $f \in D(\varphi;\psi)^A$.*

More abstractly, the composition of two partial graph homomorphisms is defined by the following commutative diagram of total graph homomorphisms. (Keep in mind that inverse images are just special pullbacks.)



Note that $D(\varphi;\psi) = D(\varphi)$ if φ is total, i.e., $H = D(\varphi)$.

Definition 5 (Order between partial graph homomorphisms)**.** *For any two parallel partial graph homomorphisms $\varphi, \phi : G \multimap H$ we have $\varphi \preceq \phi$ if, and only if, $D(\varphi) \sqsubseteq D(\phi)$ and, moreover, $\sqsubseteq;\phi = \varphi$ for the corresponding total graph homomorphisms $\varphi : D(\varphi) \to H$ and $\phi : D(\phi) \to H$.*

[1] Typing relations cannot be circular, reversed or inconsistent neither vertically, i.e., within the same hierarchy, nor horizontally, i.e., if we consider more than one hierarchy.

Typing chains appear in multilevel hierarchies as sequences of graphs from a certain graph in the hierarchy all the way up to the top of the hierarchy. They are formally defined in Definition 6.

Definition 6 (Typing Chain $\mathcal{G}$)**.** *A typing chain $\mathcal{G} = (\overline{G}, n, \tau^G)$ is given by a natural number n, a sequence $\overline{G} = [G_n, G_{n-1}, \ldots, G_1, G_0]$ of graphs of length $n+1$ and a family $\tau^G = (\tau^G_{j,i} : G_j \multimap G_i \mid n \geq j > i \geq 0)$ of partial graph homomorphisms, called **typing morphisms**, satisfying the following properties:*

- ***Total:*** *All the morphisms $\tau^G_{j,0} : G_j \to G_0$ with $n \geq j \geq 1$ are total.*
- ***Transitive:*** *For all $n \geq k > j > i \geq 0$ we have $\tau_{k,j}; \tau_{j,i} \preceq \tau_{k,i}$.*
- ***Connex:*** *For all $n \geq k > j > i \geq 0$ we have $D(\tau^G_{k,j}) \cap D(\tau^G_{k,i}) \sqsubseteq D(\tau^G_{k,j}; \tau^G_{j,i})$ and, moreover, $\tau^G_{k,j}; \tau^G_{j,i}$ and $\tau_{k,i}$ coincide on $D(\tau^G_{k,j}) \cap D(\tau^G_{k,i})$.*

Totality, transitivity and connexity ensure that for any element e in any graph G_i in a typing chain there exists a unique index m_e, with $i > m_e \geq 0$, such that e is in the domain of the typing morphism τ^G_{i,m_e} but not in the domain of any typing morphism $\tau^G_{i,j}$ with $i > j > m_e$.

Definition 7 (Individual Direct Type)**.** *For any e in a graph G_i in a typing chain $\mathcal{G} = (\overline{G}, n, \tau^G)$, with $n \geq i \geq 1$, we call $ty(e) := \tau^G_{i,m_e}(e)$ its individual direct type. We say also that e is a direct instance of $ty(e)$.*

By $df(e) = i - m_e$ we denote the difference between i and the level where $ty(e)$ is located. Usually, this difference is 1, which means that the type of e is placed at the level right above it. For convenience, we use the following abbreviations:

$$ty^2(e) = ty(ty(e)) \qquad ty^3(e) = ty(ty(ty(e))) \qquad \ldots$$
$$df^2(e) = df(e) + df(ty(e)) \qquad df^3(e) = df^2(e) + df(ty^2(e)) \qquad \ldots$$

From a general point of view, we obtain for any e in G_i a sequence of typing assignments of length $1 \leq s_e \leq$ i with $(i - df^{s_e}(e)) = 0$. The number s_e of steps depends individually on the item e. We call any of the elements $ty(e)$, $ty^2(e)$, $ty^3(e)$, ... a *transitive type* of e. The requirement that the domains of definition of typing morphisms are subgraphs ensures that for any arrow $x \xrightarrow{f} y$ in any graph G_i the non-dangling condition is satisfied: The source and the target of the direct type $ty(f) \in G_{m_f}$ of f are transitive types of x and y, respectively. Finally, we want to mention that any sequence $[G_n, G_{n-1}, \ldots, G_1, G_0]$ of graphs such that any e in any graph G_i with $n \geq i \geq 1$ has a unique individual direct type $ty(e)$ in one of the graphs $G_{i-1}, \ldots, G_1, G_0$ gives rise to a typing chain, according to Definition 6, as long as the non-dangling condition for arrows is satisfied (compare [30]).

Level 2 in Figure 1 contains instances of models described in Level 1 (called *specific-model-1* and *specific-model-2*). The nodes and references in the models depicted in Figures 1(d) and 1(e) are typed by elements defined, in this case, at

Level 1, e.g., for A1 node and G1 relation the types are A and G, respectively. At the bottom of the hierarchy (Figure 1(f)), we have (at Level 3) the *Instance* level where model *configuration-1* is displayed. Note that, even though there exists one typing chain per model (except for *Ecore*), we only focus on the typing chain computed from the bottommost level (Instance level). Notice also that in the hierarchy shown in Figure 1, the typing chain is represented by upwards dashed arrows from the instance level given by the left-hand branch of the hierarchy.

The last concept introduced in Figure 1 is *potency*, displayed as three numbers in a red box at the top right of every node, and concatenated to the name after "@" for every reference. Potencies are used on elements as a means of restricting the levels at which these elements may be used to type other elements. Thanks to potencies on elements we can define the degree of flexibility / restrictiveness we want to allow on the elements of our multilevel hierarchy. These three values are used to constrain the instantiation of elements so that the flexibility of our approach can be controlled in order to use concepts in a sensible manner. The first two values, *start* and *end*, specify the range of levels below, relative to the current one, where the element can be directly instantiated. In the example hierarchy in Figure 1, these two values are always 1, meaning that the element can only be instantiated in the level right below. A potency value of $2 - 4 - X$, for instance, would mean that an element can be directly instantiated two, three and four levels below the one where the element is defined. The third value, *depth*, is used to control the maximum number of times that the element can be transitively instantiated, regardless of the levels where this happens. That is, the amount of times an instance of that element can be re-instantiated.

In the example in Figure 1, all elements at level 1 have a depth of 2, meaning that they can be directly instantiated, and these instances can be instantiated themselves again (i.e. two times at most). This value is therefore dependent on the value of the type, and the depth of an element must always be strictly less than the depth of its type. For this reason, all elements in level 2 have a depth value of 1, and their instances of 0, meaning that they cannot be further instantiated. For elements in level 3, the instance level, the first two values also become 0, since there are no further levels below where these elements could be instantiated. In other words, the potency $0 - 0 - 0$ is used to enforce that elements at the bottom level (3) are used purely as instances, which cannot be refined further into levels below it. In general, the default potency for elements is $1 - 1 - *$ ($*$ meaning unbounded), and the potencies for all elements in the top level (Ecore) is $0 - * - *$ in order to allow, exceptionally, self-typing and to keep all instantiation initially unconstrained.

2.2. Multilevel Modelling in MultEcore - Operational semantics

Transformation rules can be used to represent actions that may happen in the system. Conventional in-place model transformations (MTs) are rule-based modifications of a source model (specified in the left-hand side of the rule) resulting in a new state of such a model (determined by its right-hand side). The left-hand side takes as input (a part of) a model and it can be understood as

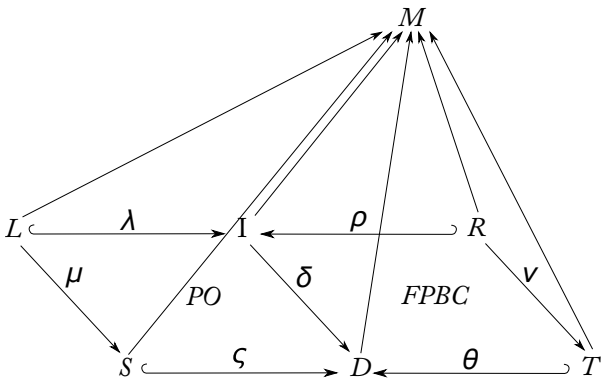


Figure 2: Conventional two-level MT rule

the pattern we want to find in our original model. The right-hand side describes the transformation we want to perform on our model and thereby the next state of the system.

Since we use graphs to formalise models, we employ graph transformation rules to express the operational semantics of multilevel models. A graph transformation rule is defined by a left L and a right R pattern. These patterns are graphs which are mapped to each other via graph morphisms λ, ρ from or to a third graph I, such that L, R, I constitute either a span ($L \longleftarrow I \longrightarrow R$) or a co-span ($L \longrightarrow I \longleftarrow R$), respectively [32, 33]. These graph morphisms are typically homomorphisms, and more specifically inclusions. Then in the span version, the graph I is the intersection of L and R, while it is the union in the co-span version. In our approach, In this paper, we use the co-span version of graph transformation rules since the graph I can be used to collect the whole context between L and R, as well as due to advantages related to the properties of the constructions used in the application of these rules [33] (see also below).

Figure 2 depicts the application of a graph transformation rule. To apply a rule ($L \hookrightarrow I \hookleftarrow R$) to a source graph S, a match μ of the left pattern in S has to be found, i.e., a graph homomorphism $\mu : L \longrightarrow S$. Then, using a pushout construction (PO), followed by a final pullback complement construction (FPBC), a target graph T will be produced [33].

We use MTs to provide definitions of behaviour by means of so-called Multilevel Coupled Model Transformations (MCMTs) [16]. MCMTs have been proposed as a means to take traditional two-level transformations rules (Figure 2) into the multilevel model world, with the right balance between precision and flexibility (see [16] for details). That is, MCMTs allow us to exploit multilevel modelling capabilities within the context of MTs. In this paper, we focus on the use of MCMTs to describe the operational semantics of DSMLs. MCMTs can also be used with other purposes, for instance, MCMTs have been used to check the structural correctness of models in [34, 27].

Figure 3 shows a simple example of an MCMT rule (called *Add and Connect*)

that models the creation of a new node and a relation between the existing node and the new one.

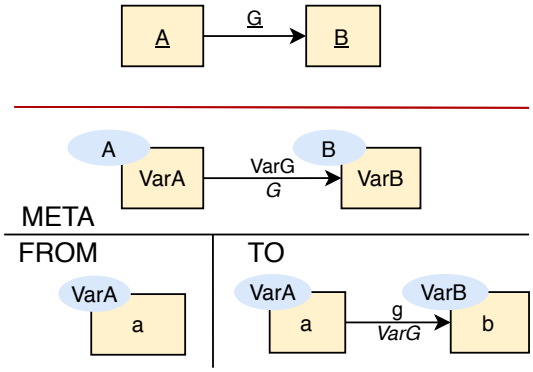


Figure 3: Rule *Add and Connect*: The execution of this rule gives a new state on the model where a new node is created and connected to the first one

The FROM and TO blocks describe the left pattern and the right pattern of the rule, respectively. The META block depicts a typing chain allowing us to locate types in any level of the chain that can be used as individual types for the items in the FROM and TO block, respectively. Notice that this is quite powerful, as META facilitates the definition of an entire multilevel pattern. At the top level of Figure 3, we mirror parts of *generic-model-1*, defining elements like A, B and G as constants. We differentiate constants as their names are underlined and their types are not specified via the ellipse above (for nodes) or the italic text (for references). The use of constants constrains the matching process, significantly reducing the amount of matches. The rule can be applied to models (instances) typed by the left-hand typing chain of Figure 1 (i.e., *specific-model-1, generic-model-1, Ecore*).

Note, that the horizontal lines do not enforce consecutiveness between the levels specified in the rule with respect to the hierarchy. This leads to a more natural way of defining that a type is defined at some level above, without explicitly stating in which level. In fact, this also promotes flexibility in case of future modifications of the number of branches (horizontal dimension) and the depth (vertical dimension) of hierarchies. Consider for the horizontal dimension, for instance, in the example in Figure 1, adding a new model called *specific-model-1'*, branching at level 2, as instance of *generic-model-1*. For the vertical dimension, consider for example introducing a new level between levels 2 and 3 to create a more refined model (called, e.g., *more-specific-model-1*). The key aspect is that none of these extensions would require the modification of other models in the hierarchy, nor the rule depicted in Figure 3, while the MCMT would still be valid. This flexibility is achieved as we allow the types on the variables to be transitive types. For instance, VarA (placed at the second level of the META), typed by the variable A, would match any node which indirectly has A as type, or ultimately will match to A if no indirect one is found. A correct

match of the rule comes when an element, coupled together with its type, fits an instance of VarA (e.g., a located in the FROM part).

Given the current state of the hierarchy in Figure 1, any instances of elements matching the pattern VarA would be candidates to perform the transformation. This in turn makes it possible to apply the rule to either instances of A1 or to instances of A2 (these elements are defined in model *specific-model-1* at Figure 1).

The general structure of an MCMT and its application is displayed in Figure 4. The figure can be visualised as two flat trees, each of them defined by typing chains and connected to each other by matching morphisms.

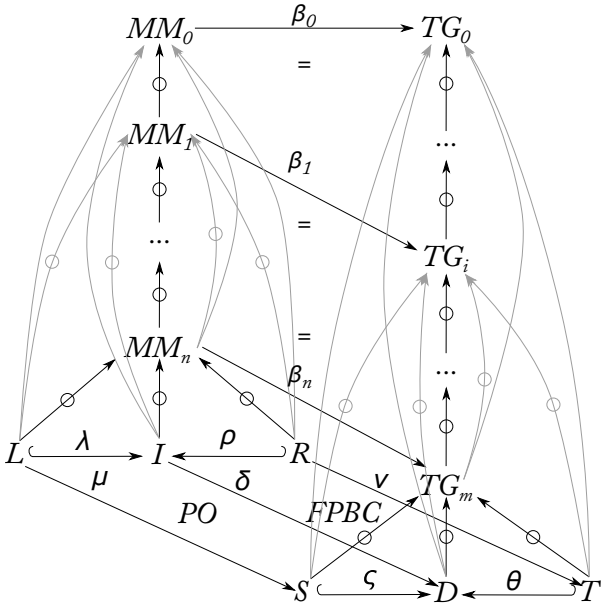


Figure 4: Formal construction for MCMT

The tree on the left contains the pattern that the user defines in the rule. It consists of the left and right parts of the rule (FROM and TO, respectively), represented as L and R in the diagram, and the interface I that is the union of both L and R, being λ and ρ inclusion graph homomorphisms.

These three graphs are typed by elements in the same typing chain $\mathcal{MM} = (\overline{MM}, n, \tau^{MM})$, defined in the META block, which is depicted as a sequence of metamodels MM_i, for $0 \leq i \leq n$, that ends with the root of the chain MM_0 (Ecore in our case). The multilevel typing of the graphs L, I, R is given by families of typing morphisms.

Definition 8 (Multilevel Typed Graph)**.** *A* ***multilevel typed graph*** (H, σ^H) *is a graph H with a* ***multilevel typing*** $\sigma^H : H \Rightarrow \mathcal{G}$ *of H over a typing chain $\mathcal{G} = (\overline{G}, n, \tau^G)$ given by a family $\sigma^H = (\sigma^H : H \multimap G_i \,|\, n \geq i \geq 0)$ of partial graph homomorphisms.*

So, the rule is given by multilevel typed graphs $(L, \sigma^L : L \Rightarrow \mathcal{MM})$, $(R, \sigma^R : R \Rightarrow \mathcal{MM})$, $(I, \sigma^I : I \Rightarrow \mathcal{MM})$ with $I = L \cup R$ such that σ^L and σ^R coincide on the intersection $L \cap R$ and σ^I is constructed as the union of σ^L and σ^R.

For the rule to be applied, we have to find a match (a graph homomorphism μ) of the pattern graph L into an instance graph S at the bottom of the current application hierarchy. The choice of S determines a sequence $[S, TG_m, TG_{m-1}, \ldots, TG_1, TG_0]$ of graphs from S up to the top of the hierarchy. The sequence $[TG_m, TG_{m-1}, \ldots, TG_1, TG_0]$ of graphs constitutes a typing chain $\mathcal{TG} = (\overline{TG}, m, \tau^{TG})$ and the family of typing morphisms from S into TG_i, $m \geq i \geq 0$ turns S into a multilevel typed graph $(S, \sigma^S : S \Rightarrow \mathcal{TG})$. The match $\mu : L \to S$ has, however, to satisfy some application conditions: There has to be a match of the typing chain $\mathcal{MM}$ into the typing chain $\mathcal{TG}$ that is compatible with the multilevel typings σ^L, σ^S and the match μ. Matches of typing chains are described by a very flexible concept of morphisms between typing chains.

Definition 9. *A **typing chain morphism** $(\phi, f) : \mathcal{G} \to \mathcal{H}$ between two typing chains $\mathcal{G} = (\overline{G}, n, \tau^G)$ and $\mathcal{H} = (\overline{H}, m, \tau^H)$ with $n \leq m$ is given by*

- *a function $f : [n] \to [m]$, where $[n] = \{0, 1, 2, \ldots, n\}$, such that $f(0) = 0$ and $j > i$ implies $f(j) > f(i)$ for all $i, j \in [n]$, and*
- *a family of total graph homomorphisms $\phi = (\phi_i : G_i \to H_{f(i)} \mid i \in [n])$ with*

$$\tau^G_{j,i}; \phi_i \leq \phi_j; \tau^H_{f(j),f(i)} \quad \text{for all } n \geq j > i \geq 0. \tag{1}$$

*A typing chain morphism $(\phi, f) : \mathcal{G} \to \mathcal{H}$ is called **closed** if, and only if, $\tau^G_{j,i}; \phi_i = \phi_j; \tau^H_{f(j),f(i)}$ for all $n \geq j > i \geq 0$.*

There are three flexibility features we want to underline: (1) Jumps of typing can be arbitrarily stretched in the sense, that the difference $f(j) - f(i)$ can be bigger than the difference $j - i$. (2) We require, in general, only that typing is preserved, i.e., if an element $\mathtt{e}$ in G_j has a transitive type in G_i then the image $\phi_j(\mathtt{e})$ in $H_{f(j)}$ is required to have a transitive type in $H_{f(i)}$. For closed typing chain morphisms, we require, however, that typing is also reflected, i.e., if the image $\phi_j(\mathtt{e})$ in $H_{f(j)}$ has a transitive type in $H_{f(i)}$ it is required that $\mathtt{e}$ has a transitive type in G_i. (3) The granularity of typing does not need to be preserved, i.e., if an element $\mathtt{e}$ in G_j has a direct (!) type in G_i then the image $\phi_j(\mathtt{e})$ in $H_{f(j)}$ needs only to have a transitive type in $H_{f(i)}$.

The graph homomorphisms $\beta_n, \ldots, \beta_1, \beta_0$ and the assignments $0 \mapsto 0, 1 \mapsto i, \ldots, n \mapsto m$ in Figure 4 depict the required typing chain morphism (match) $(\beta, f) : \mathcal{MM} \to \mathcal{TG}$. To describe type compatibility of matches and the result of an MCMT application we need to have the composition of typing chain morphisms at hand.

Definition 10 (Composition of typing chain morphisms)**.** *The composition $(\phi, f); (\psi, g) : \mathcal{G} \to \mathcal{K}$ of two typing chain morphisms $(\phi, f) : \mathcal{G} \to \mathcal{H}$, $(\psi, g) :$*

$\mathcal{H} \to \mathcal{K}$ *between typing chains* $\mathcal{G} = (\overline{G}, n, \tau^G)$, $\mathcal{H} = (\overline{H}, m, \tau^H)$, $\mathcal{K} = (\overline{K}, l, \tau^K)$ *with* $n \leq m \leq l$ *is defined by*

$$(\phi, f); (\psi, g) := (\phi; \psi_{\downarrow f}, f; g)$$

where $\psi_{\downarrow f} := (\psi_{f(i)} : H_{f(i)} \to K_{g(f(i))} \mid i \in [n])$ *and thus*

$$\phi; \psi_{\downarrow f} := (\phi_i; \psi_{f(i)} : G_i \to K_{g(f(i))} \mid i \in [n]).$$

Chain denotes the category of typing chains and typing chain morphisms.

It turns out that multilevel typings are not appropriate to formulate adequate compatibility conditions for matches. Therefore, we describe multilevel typing by means of inclusion chains and typing chain morphisms.

Lemma 1 (Inclusion chain)**.** *For any graph* H *we can extend any sequence* $\overline{H} = [H_n, H_{n-1}, \ldots, H_1, H_0]$ *of subgraphs of* H*, with* $H_0 = H$*, to a typing chain* $\mathcal{H} = (\overline{H}, n, \tau^H)$ *where for all* $n \geq j > i \geq 0$ *the corresponding* ***partial inclusion graph homomorphism*** $\tau^H_{j,i} : H_j \multimap H_i$ *is given by* $D(\tau^H_{j,i}) := H_j \cap H_i$ *and the span of total inclusion graph homomorphisms*

$$H_j \xhookleftarrow{\sqsubseteq} D(\tau^H_{j,i}) = H_j \cap H_i \xhookrightarrow{\tau^H_{j,i}} H_i$$

By means of Lemma 1, we can represent now the four given multilevel typings $\sigma^L : L \Rightarrow \mathcal{MM}$, $\sigma^I : I \Rightarrow \mathcal{MM}$, $\sigma^R : R \Rightarrow \mathcal{MM}$, and $\sigma^S : S \Rightarrow \mathcal{TG}$, equivalently, by four corresponding inclusion chains (see Figures 5 and 6)

- $\mathcal{L} = (\overline{L}, n, \tau^L)$ with $L_i := D(\sigma^L_i)$ for all $i \in [n]$ and thus $L_0 = L$,
- $\mathcal{I} = (\overline{I}, n, \tau^I)$ with $I_i := D(\sigma^I_i)$ for all $i \in [n]$ and thus $I_0 = I$,
- $\mathcal{R} = (\overline{R}, n, \tau^R)$ with $R_i := D(\sigma^R_i)$ for all $i \in [n]$ and thus $R_0 = R$ and
- $\mathcal{S} = (\overline{S}, m, \tau^S)$ with $S_j := D(\sigma^S_j)$ for all $j \in [m]$ and thus $S_0 = S$,

together with four typing chain morphisms

- $(\sigma^L, id_{[n]}) : \mathcal{L} \to \mathcal{MM}$ with $\sigma^L = (\sigma^L_i : L_i \to MM_i \mid i \in [n])$,
- $(\sigma^I, id_{[n]}) : \mathcal{I} \to \mathcal{MM}$ with $\sigma^I = (\sigma^I_i : I_i \to MM_i \mid i \in [n])$,
- $(\sigma^R, id_{[n]}) : \mathcal{R} \to \mathcal{MM}$ with $\sigma^R = (\sigma^R_i : R_i \to MM_i \mid i \in [n])$, and
- $(\sigma^S, id_{[m]}) : \mathcal{S} \to \mathcal{TG}$ with $\sigma^S = (\sigma^S_j : S_j \to TG_j \mid j \in [m])$.

By construction, we have $I_i = L_i \cup R_i$ for all $i \in [n]$ thus the family of inclusion graph homomorphisms $\lambda_i : L_i \hookrightarrow I_i$, $i \in [n]$ establishes a closed typing chain morphism $(\lambda, id_{[n]}) : \mathcal{L} \to \mathcal{I}$ while the family of inclusion graph homomorphisms $\rho_i : R_i \hookrightarrow I_i$, $i \in [n]$ establishes a closed typing chain morphism $(\rho, id_{[n]}) : \mathcal{R} \to \mathcal{I}$. Finally, the construction of $\mathcal{I}$ ensures type compatibility of the rule:

$$(\lambda, id_{[n]}); (\sigma^I, id_{[n]}) = (\sigma^L, id_{[n]}) \quad \text{and} \quad (\rho, id_{[n]}); (\sigma^I, id_{[n]}) = (\sigma^R, id_{[n]}) \quad (2)$$

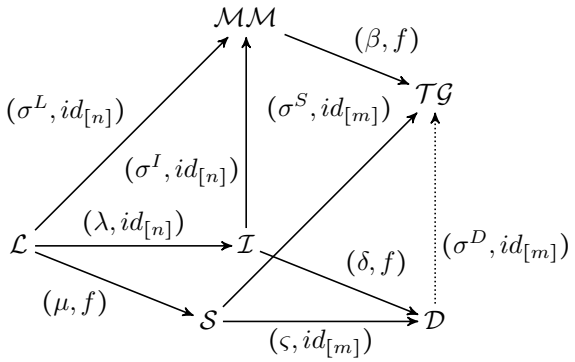


Figure 5: Pushout step

Type compatibility of the matches $\mu : L \to S$ and $(\beta, f) : \mathcal{MM} \to \mathcal{TG}$ means that $\mu : L \to S$ restricts for each $i \in [n]$ to a map $\mu_i : L_i \to S_{f(i)}$ such that this family of graph homomorphisms establishes a typing chain morphism $(\mu, f) : \mathcal{L} \to \mathcal{S}$ satisfying the equation

$$(\sigma^L, id_{[n]}); (\beta, f) = (\mu, f); (\sigma^S, id_{[m]}). \tag{3}$$

The type compatibility requirements for rules and matches ensure that the pushout for graphs, at the bottom of Figure 4, gives rise to a pushout for the corresponding inclusion chains at the bottom of Figure 5: For each $n \geq i > 0$ we set $D_{f(i)} := S_{f(i)} \cup \delta(I_i)$ thus the co-span $S \overset{\varsigma}{\hookrightarrow} D \overset{\delta}{\hookleftarrow} I$ restricts to a co-span $S_{f(i)} \overset{\varsigma_{f(i)}}{\hookrightarrow} D_{f(i)} \overset{\delta_i}{\hookleftarrow} I_i$. This co-span can be proven to be a pushout of the span $S_{f(i)} \overset{\mu_i}{\hookleftarrow} L_i \overset{\lambda_i}{\hookrightarrow} I_i$. To get a complete inclusion chain $\mathcal{D}$ of length m, we simply set $D_j := S_j$ and $\varsigma_j := id_{S_j}$ for all $j \in [m] \setminus f([n])$. The complex proof that this simple construction provides indeed a pushout in **Chain** can be found in [30].

Since the bottom square in Figure 5 is a pushout, the type compatibily conditions (2) and (3) ensure that there is a unique typing chain morphism $(\sigma^D, id_{[m]})$ from $\mathcal{D}$ to $\mathcal{TG}$ such that

$$(\delta, f); (\sigma^D, id_{[m]}) = (\sigma^I, id_{[n]}); (\beta, f)\ ,\ (\varsigma, id_{[m]}); (\sigma^D, id_{[m]}) = (\sigma^S, id_{[m]}) \tag{4}$$

This shows, that we have indeed constructed a type compatible multilevel typing of the graph D.

For the second step of rule application, namely the FPBC construction shown in Figure 6, we first construct FPBC in category **Graph** and obtain T. It will remain to reconstruct the typing of T in order to create an inclusion chain $\mathcal{T} = (\overline{T}, m, \tau^T)$. To achieve this, we construct the reduct of $\mathcal{D} = (\overline{D}, m, \tau^D)$ along $\theta : T \hookrightarrow D$ and $id_{[m]}$ by level-wise intersection (pullback) for all $n \geq i \geq 1$. In such a way, we obtain an inclusion chain $\mathcal{T} = (\overline{T}, m, \tau^T)$ together with a closed

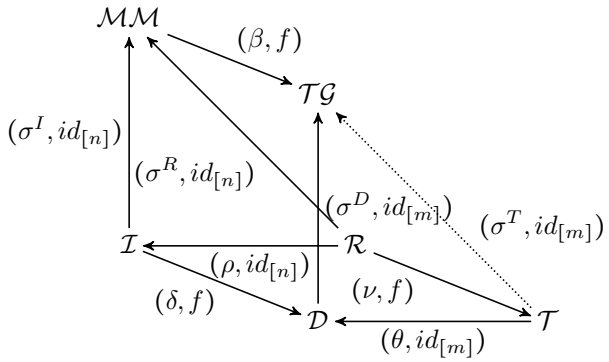


Figure 6: Final pullback complement step

typing chain morphism $(\theta, id_{[m]}) : \mathcal{T} \to \mathcal{D}$. The multilevel typing of $\mathcal{T}$ is simply borrowed from $\mathcal{D}$, that is, we define (see Fig. 6)

$$(\sigma^T, id_{[m]}) := (\theta, id_{[m]}); (\sigma^D, id_{[m]}) \tag{5}$$

and this trivially gives us the intended type compatibility of $(\theta, id_{[m]})$.

The specific conditions that are required are out of the scope of this paper, and can be consulted in the technical report [30].

3. Composition

In current MDSE practice, DSMLs are built by language designers using a metamodel defined by a general-purpose meta-modelling language [35], like MOF. As mentioned in Section 1, this in turn leads to a metamodel that describes the instances that users of the language can build in the immediate metalevel below. Thus, languages are specified within two levels: definition and usage. However, the increasing complexity of software systems advocates the need for more DSMLs as refinement of general-purpose languages [36]. Hence, the need for alternative techniques that alleviate the two-level restrictions (provided, for instance, by MLM) becomes progressively significant.

By using MLM capabilities, one could customise families of similar DSMLs, where certain commonalities are shared. In this context, the challenge for language designers is to take advantage of the existing commonalities among similar DSMLs by reusing, as much as possible, formerly defined language constructs [2]. Furthermore, having a way to modularise a language to create *features* — to later reuse and combine them — can be used in different manners to produce tailor-made DSMLs targeting the needs of well-defined audiences. This feature-oriented approach to DSML engineering requires the definition of DSMLs in a modularised fashion where language features are implemented as interdependent and composable language modules.

3.1. Standard Composition Approach

A consequence of having DSMLs that tackle scoped problem spaces (enhancing separation of concerns), is that often we find ourselves thinking that one of them is not enough to reason about certain global properties or to execute the complete system. In other words, it might be necessary to compose some of the constructed models to achieve such goals. In general, model composition unfolds along two dimensions, structure and behaviour.

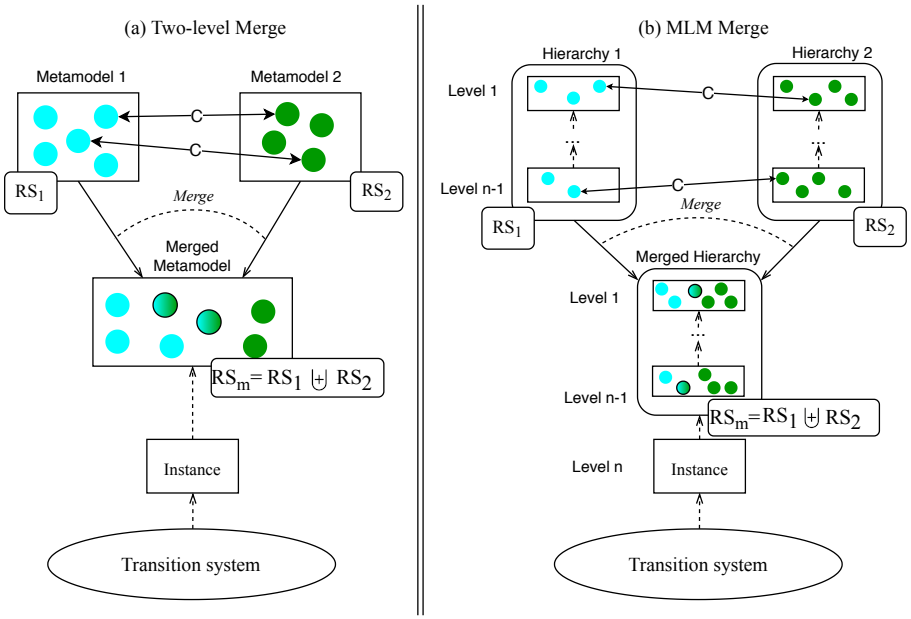


Figure 7: Two-level merge combination vs MLM merge combination

Commonly, frameworks that offer composition operators had to define their own composition rules and provide custom-made implementations of such operators (e.g., through model transformations). To alleviate ad-hoc implementations and to provide standard operations, several researchers have proposed in [19] a paradigmatic *merging* operation for structure composition and *event scheduling* for behavioural composition. Intuitively, merging refers to the operation in which "the common elements are included only once, while the rest are preserved". Figure 7(a) shows how the *merge* operation in [19] works for two level approaches. Formally, a merge combination operator takes two metamodels, *Metamodel 1* and *Metamodel 2* as inputs, as well as a set of correspondence tuples $C = \{\langle e_x, e_y \rangle, \ldots\}$ with $e_x \in$ *Metamodel 1* and $e_y \in$ *Metamodel 2*. The merge combination operator produces a new output *Merged Metamodel* that contains, for each tuple $\langle e_x, e_y \rangle \in C$, a single metamodel element. All metamodel elements in *Metamodel 1* and *Metamodel 2* that are not given a correspondence in C are simply copied into the *Merged Metamodel*. In fact, this common and standard

representation of merging is a colimit construction and goes back to Burstall and Goguen's work in the late 70's [37].

Note that the elements displayed as circles in either of the metamodels, are just abstract representations and could be a node or a reference; they are displayed in this way to show how the combination is done after identifying corresponding elements. To represent each of the merged elements we use gradient colour surrounded by line, which represent combined elements originally coming from two individual ones. Also, the merge operator could take more than two metamodels as inputs, as long as the set of correspondences C is properly specified [38], but we discuss here the case of two for simplification purposes.

RS_1 and RS_2 are the sets of transformation rules attached to *Metamodel 1* and *Metamodel 2*, respectively. In the same way, as we obtain a *Merged Metamodel* by the merge operation, a set of rules (RS_m) to be attached to such a merged metamodel is produced by the disjoint union of each of the rule sets ($\mathsf{RS}_\mathrm{m} = \mathsf{RS}_1 \uplus \mathsf{RS}_2$).

Instance models can be then specified by defining elements that are typed (recall that dashed arrows represent typing graph morphisms) by elements located in the *Merged Metamodel*. These instances can be executed producing the *Transition system* (state space) which is obtained by applying the rules that come from the resulting rule set RS_m.

If we apply the merging approach to the MLM case, we get the situation depicted in Figure 7(b). Following the same approach as for the two-level case, we merge two multilevel hierarchies, *Hierarchy 1* and *Hierarchy 2*, for which the merging process would be done level-wise. If there exists some level mismatch between the hierarchies, one can still establish correspondences among elements, however, the resulting *Merged Hierarchy* must be structurally correct and fulfil the corresponding multilevel constraints. The degree of safeness of the different proposals implementing this approach depends on the amount of *sanity checks* in each of them [31]. As stated at the beginning of Section 2.1, in our implementation we provide mechanisms to assure that potency on elements is preserved, and typings are correctly applied.

Shortcomings of the standard merge operator. A crucial shortcoming present in the merge composition approach is the loss of the "individuality" nature of the merged elements (see also [39] for further shortcomings related to constraint checking). This means that the original elements that have been merged into a new one cannot be used separately after the merge. This capability might be useful in several situations. For example, when the elements about to be composed are not identical, but powering up each other. In these situations, we may need to use in our models the merged elements when we want to take advantage of all the features each of them provides. However, certain parts of the model might require their isolated aspects (i.e., the original, separated elements) to be available. These merged elements are no longer available as individual elements of the metamodel and hence cannot be instantiated at the *Instance* level.

3.2. Composition of hierarchies in MultEcore

Our proposal is to provide elements with multiple *natures*. Natures can be dynamically added and removed, so elements can have their own specific features, while still being able to define a combined and enriched nature. Our formalisation of typing chains allows us to incorporate or remove additional natures, as types, to elements. For instance, given a situation where we are working with two typing chains, each of our nodes and references residing at the instance level would be double-typed, each one provided by each of the typing chains. But also, at any time, a typing chain can be removed without affecting the other. Elements can therefore have, simultaneously, as many types as we need. This can be seen as an aspect-like mechanism that we can use as we require, being able to use aspects independently or together. The same principles apply to the definitions of behaviour by the amalgamation of MCMTs (Section 3.2.2). The fact that typing chains may be added and removed as needed makes the composition of DSMLs very flexible.

3.2.1. Composition of multilevel modelling hierarchies

Our MLM approach does not restrict the number of typing chains that can be specified in a hierarchy. Frequently, we denote a multilevel hierarchy as the *main* or *default* one and call it *application hierarchy*, since it represents the main language being designed. An application hierarchy can optionally include an arbitrary number of *supplementary hierarchies* which add new aspects to the application one. Note that we distinguish the typing chains and individual typing relations of the application hierarchy with blue colours, and use green for the supplementary ones. Adding or removing supplementary hierarchies is made possible by the incorporation or extraction of additional typing chains. For instance, we might have different hierarchies (physically separated, e.g., different projects in the MultEcore tool) that we want to compose. Such a result can be achieved by assigning the role of application hierarchy to one of them and adding the rest as supplementary ones. These two different "roles" assigned to hierarchies are used for the most part in this paper, since it facilitates the reusability and the modularisation of the system being modelled. However, it is important to point out that, as long as the typing chains are properly defined and consistent, the formalisation of application and supplementary typing chains has no real difference. Therefore, we can consider both working with several hierarchies, for which there might be several *Ecore* models at the top, or with several branches within the same hierarchy where there is only one *Ecore* model. The latter alternative can be achieved using the same techniques as the former, as long as some of our constraints are weakened, e.g., the tree shape (discussed in Section 2.1) that we impose on hierarchies or the single individual type (Definition 7) of each element in a hierarchy.

Figure 8 displays the hierarchy in Figure 1, but in it two different branches are combined within the same hierarchy, i.e., we specify two typing chains. The left-hand branch, in which models are connected by blue dashed arrows, represents the main typing chain and guides how we can consistently and precisely

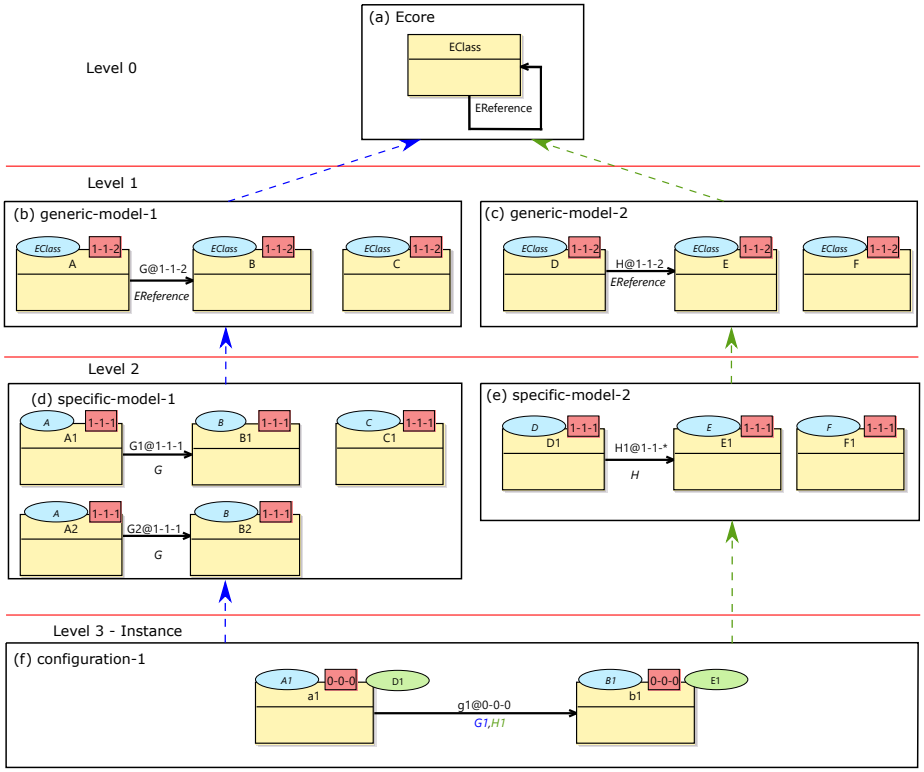


Figure 8: Multilevel hierarchy with two typing chains

type elements. As described above, we can then add extra typing chains, in this case, to our instance level, for example the one represented by the green dashed arrows (characterised by the right-hand branch). Once a new typing chain is incorporated, all the elements (both nodes and references) need to be extended with a new type. Then, these types can be used/modified by the modeller as it is done with the main type.

The model configuration-1 in Fig. 8(f) shows an example of how elements may be double-typed. One can see that node a1 has two types associated, A1 from the left-hand typing chain, its main type, and D1 from the right-hand typing chain, which adds additional information to the node. We have a similar situation with reference g1 and its two types G1 and H1.

Figure 9 compares the merge case exposed in Figure 7(b) with our approach for composition based on multiple typing chains. As already explained, a considerable drawback of the merge operation is that, once the merge is performed, the individuality of the elements that belonged to the different models prior the composition step is lost. Notice in Figure 9(b) that we do not carry any "physical" merge when a composed hierarchy or model is produced, but we can instantiate elements with more than one type. The hierarchies are left un-

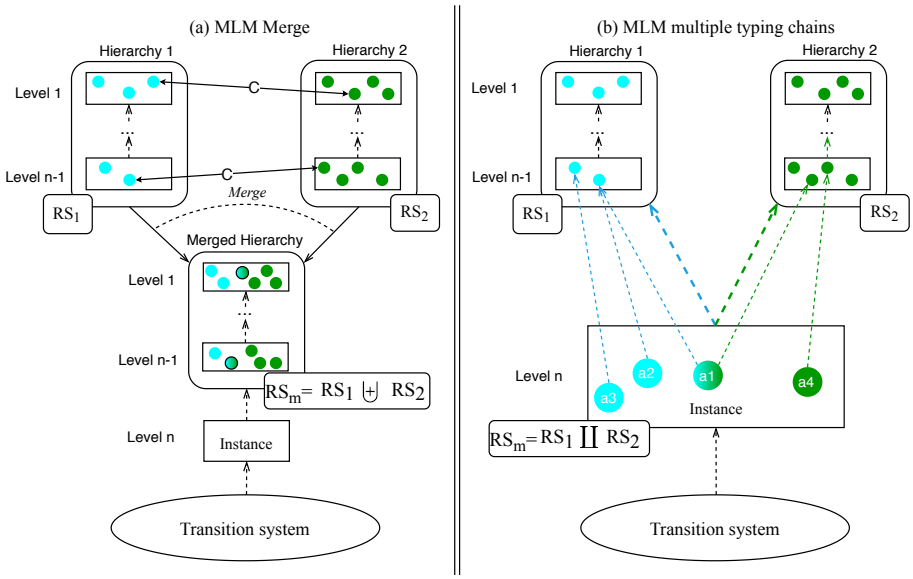


Figure 9: MLM merge combination vs our approach with multiple typing chains

touched, but the rules belonging to each hierarchy might be amalgamated to take into account a desired composed behaviour. Of course, we can preserve the "individual" nature by using just one of the types as shown in either a2, a3 or a4 elements in Figure 9(b). We discuss in section 3.2.2 how we achieve behaviour composition ($\mathsf{RS_m} = \mathsf{RS_1} \coprod \mathsf{RS_2}$ at the bottom of Figure 9(b)).

The inclusion of an extra typing chain forces all the elements at the instance level to have an additional new type from the newly incorporated typing chain. Elements which do not get a specific type from the newly added typing chain will get a default typing; i.e., the type of the nodes is set to EClass (and arrows to EReference, respectively). Recall that this default typing to Ecore elements is independent on whether the new typing chain is contained in the same hierarchy (i.e., we use the same Ecore as a top most model G_0) or we use a completely new hierarchy. This is illustrated in Figure 10 which depicts a fragment of the hierarchy of Figure 9(b) but using typing arrows (formally $ty(e)$) instead of ellipses. We can see that the model *configuration-1* has two typing chains: a blue and a green one, in the same hierarchy. Note that, in a particular typing chain we omit the default typing to Ecore elements if other intermediate types exist (e.g., a2 has A1 in the blue branch, while it has only the default EClass in the green one). We describe the individual typing for each of the elements below; we denote $TC_x(e)$, with $x = 1, 2$, the corresponding individual typings of

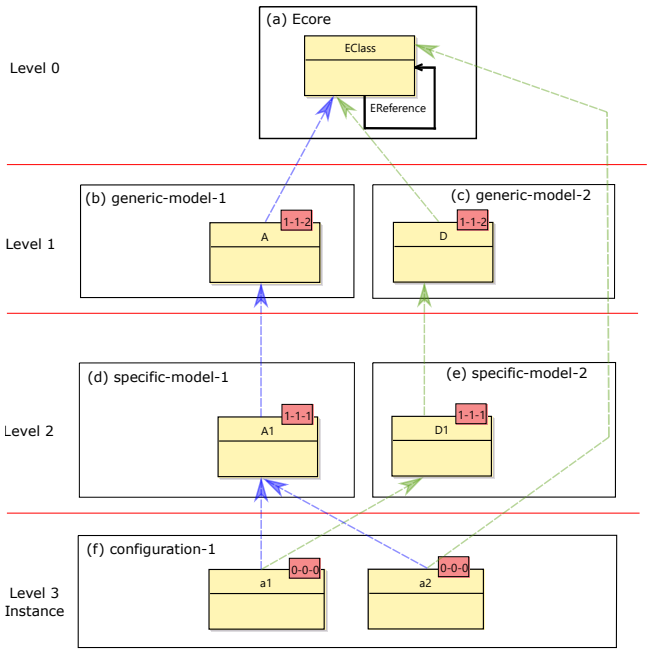


Figure 10: Typing chains to keep individuality of elements

the element e in typing chain x:

$$TC_1(a1) \equiv a1 \;\longmapsto\; ty_1(a1) = A1 \;\longmapsto ty_1^2(a1) = A \;\longmapsto ty_1^3(a1) = EClass$$
$$TC_2(a1) \equiv a1 \;\longmapsto\; ty_2(a1) = D1 \;\longmapsto ty_2^2(a1) = D \;\longmapsto ty_2^3(a1) = EClass$$

$$TC_1(a2) \equiv a2 \;\longmapsto\; ty_1(a2) = A1 \;\longmapsto ty_1^2(a2) = A \;\longmapsto ty_1^3(a2) = EClass$$
$$TC_2(a2) \equiv a2 \;\longmapsto\; ty_2(a2) = EClass$$

3.2.2. Amalgamation of MCMTs

In the previous section, we explained how we support composition of MLM models by multiple typing. In this section, we will explain composition of behaviour by the amalgamation of MCMT rules. The amalgamation of transformation rules has been widely discussed in the literature in the context of traditional (two-level) approaches [40, 41, 42, 43, 44]. In this paper, we study amalgamation in the MLM context and allow potentially conflicting rules to be amalgamated under certain constraints.

We are working with two MLM hierarchies or, as in the running example, with the composition of the two branches of Figure 8, each of them with its own set of MCMTs. The elements will appear double-typed at the instance level (for example, the situation described in Figure 8(f)). Thus, a key aspect is to also be able to amalgamate rules which only pertain to each branch of the hierarchy.

To illustrate the constructions, we will explain the process by amalgamating two MCMTs, one for each branch: Rule A (TR_A) for the left branch, which is the rule depicted in Figure 3 and shown again in Figure 11(a), together with Rule B (TR_B), which is a very similar rule for the right branch (Figure 11(b)).

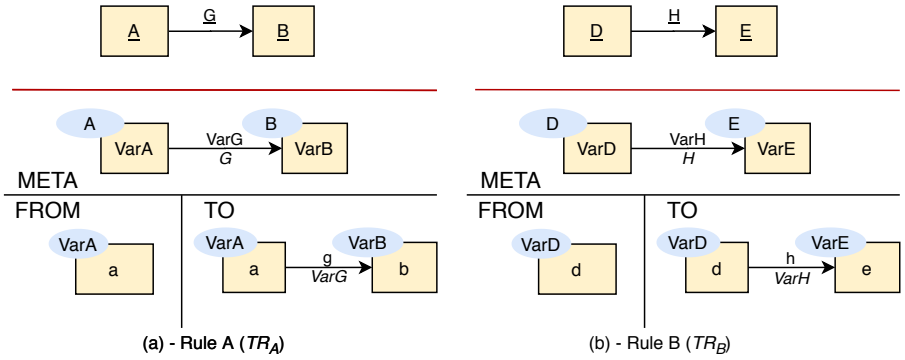
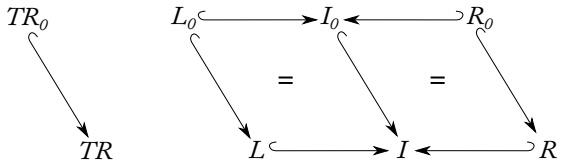


Figure 11: MCMT rules to be amalgamated: (a) Rule A affecting the left-hand branch and (b) Rule B affecting the right-hand branch

An essential step to achieve amalgamation (or, in general, composition) is the identification process where the elements that correspond to each other have to be identified. Most works in the literature use a so-called kernel rule to express correspondences between two or more rules [40, 41, 42, 43]. Also in our approach, we assume that the user provides the correspondences between elements in the rules which are to be amalgamated. That is, given Rule A $L_A \hookrightarrow I_A \hookleftarrow R_A$ and Rule B $L_B \hookrightarrow I_B \hookleftarrow R_B$, the correspondences provided by the user (L_0, I_0 and R_0) will be defined as a subrule TR_0 such that $TR_0 \hookrightarrow TR_A$ and $TR_0 \hookrightarrow TR_B$.

Definition 11 (Subrule)**.** *A rule $TR_0 := L_0 \hookrightarrow I_0 \hookleftarrow R_0$ is a subrule of a rule $TR := L \hookrightarrow I \hookleftarrow R$, written $TR_0 \hookrightarrow TR$, where there exist three inclusion graph morphisms $L_0 \hookrightarrow L$, $I_0 \hookrightarrow I$, and $R_0 \hookrightarrow R$, such that the following diagrams are commutative.*



This will give rise to three spans with inclusion graph morphisms: $L_A \hookleftarrow L_0 \hookrightarrow L_B$, $I_A \hookleftarrow I_0 \hookrightarrow I_B$ and $R_A \hookleftarrow R_0 \hookrightarrow R_B$. Recall that $L \sqsubseteq I$ and $R \sqsubseteq I$, hence, we can deduce R_0 and L_0 from I_0, meaning that in practise the user only needs to specify I_0.

Amalgamating TR_A and TR_B w.r.t. TR_0 means to combine the components of the rules so that we obtain a single rule TR_M such that $(L_M = L_A +_{L_0} L_B) \hookrightarrow$

$(I_M = I_A +_{I_0} I_B) \leftarrow (R_M = R_A +_{R_0} R_B)$. Again, we use pushout constructions, as a common practise, to obtain the components of TR_M. Below, we detail the construction of L_M as the pushout $L_A +_{L_0} L_B$ (see Figure 12). The same constructions will apply for the I and R components of the rules.

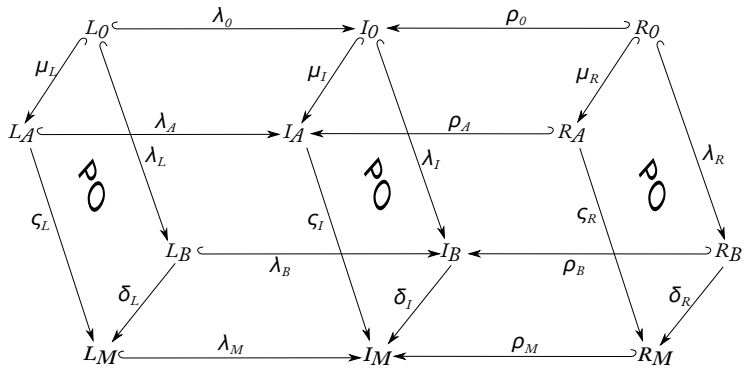


Figure 12: Amalgamated rule construction with pushouts

However, since L_A and L_B (and, respectively, I_A, I_B, R_A and R_B) have different multilevel types, we would need to unify the types by defining default types for each of the elements in the other hierarchies, i.e., all L_A elements would have the default type (EClass/EReference) from the typing chain $\mathcal{MM}_\mathcal{B} = (\overline{MM_B}, n_b, \tau^{MM_B})$, while all L_B elements will have the default types from the typing chain $\mathcal{MM}_\mathcal{A} = (\overline{MM_A}, n_a, \tau^{MM_A})$ (and again, the same for I_A, I_B, R_A and R_B). Furthermore, L_0 would have the default types in both chains.

We illustrate this in Figure 13. On the left-hand side, we break down the $L_A +_{L_0} L_B$ pushout resulting in L_M together with their respective typing chains. As described above, a is typed over $\mathcal{MM}_\mathcal{A}$ (VarA, A, EClass) and over $\mathcal{MM}_\mathcal{B}$ (EClass), d is typed over $\mathcal{MM}_\mathcal{B}$ (VarD, D, EClass) and by EClass over $\mathcal{MM}_\mathcal{A}$, a ≡ d in L_0 is only double-typed by EClass in each of the typing chains, and the resulting ad in L_M is typed over $\mathcal{MM}_\mathcal{A}$ (via $\sigma^{L^A_M}$) and $\mathcal{MM}_\mathcal{B}$ (via $\sigma^{L^B_M}$) as shown in the right-hand side of Figure 13.

Expressed in terms of inclusion chains, the aforementioned typing relations mean that $L^A_{0,0} = L^B_{0,0} = L_{0,0} = L_0$, where $L^A_{0,0}$ is the part of L_0 which is typed by $MM_{A,0}$ (see Lemma 1). The rest of the levels $L^A_{0,i}$, with $0 < i \leq n_a$ will be empty since L_0 has only default types in the two rules' hierarchies. These types are reflected by the two light thin arrows from L_0 to the two EClasses in $\mathcal{MM}_\mathcal{A}$ and $\mathcal{MM}_\mathcal{B}$ in Figure 13, respectively. Similarly, we have $L^A_{A,0} = L^B_{A,0} = L_{A,0}$ and $L^A_{B,0} = L^B_{B,0} = L_{B,0}$. The levels (except for 0) of the inclusion chains $\mathcal{L}_\mathcal{A}$ (resp. $\mathcal{L}_\mathcal{B}$) along $\sigma^{L^A_A} : L_A \Rightarrow \mathcal{MM}_\mathcal{A}$ (resp. $\sigma^{L^B_B} : L_B \Rightarrow \mathcal{MM}_\mathcal{B}$) will be constructed according to Lemma 1. Moreover, the default levels (except for 0) of the inclusion chains $\mathcal{L}_\mathcal{A}$ (resp. $\mathcal{L}_\mathcal{B}$) along $\sigma^{L^B_A} : L_A \Rightarrow \mathcal{MM}_\mathcal{B}$ (resp. $\sigma^{L^A_B} : L_B \Rightarrow \mathcal{MM}_\mathcal{A}$) will be empty. Having these typing chains, we apply level-wise pushouts as described in Section 2.2 [30].

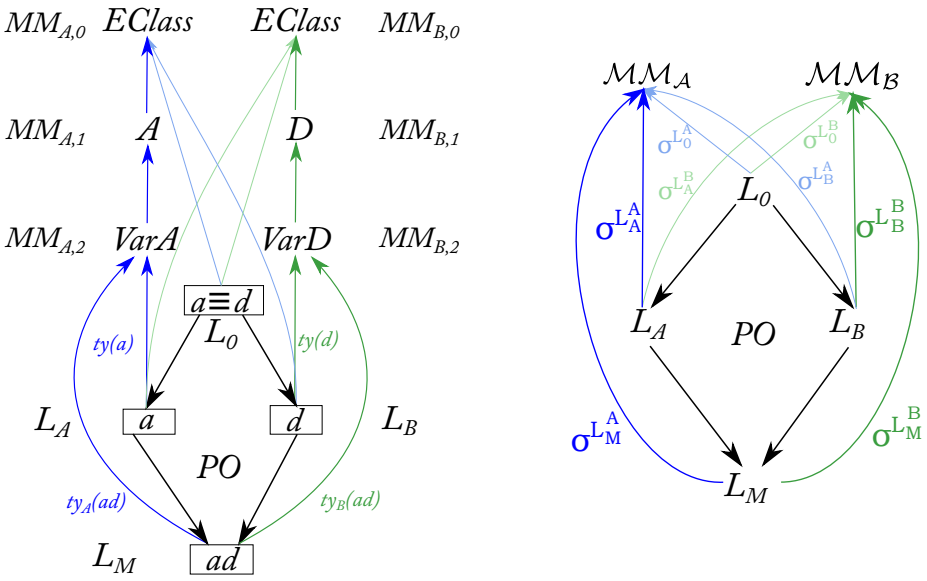


Figure 13: L_M with typing chains as result of pushout of L_A, L_B modulo L_0

The results of these level-wise pushouts would be two inclusion chains:

- $\mathcal{L}^{\mathcal{A}}_{\mathcal{M}} = (\overline{L^A_M}, n_a, \tau^{L^A_M})$ with $\sigma^{L^A_M} : L^A_M \Rightarrow \mathcal{MM}_{\mathcal{A}}$: The levels $L^A_{M,i}$ for all $0 \leq i \leq n_a$ of the inclusion chain will be produced by the pushouts of the spans $L^A_{A,i} \hookleftarrow L^A_{0,i} \hookrightarrow L^A_{B,i}$.
- $\mathcal{L}^{\mathcal{B}}_{\mathcal{M}} = (\overline{L^B_M}, n_b, \tau^{L^B_M})$ with $\sigma^{L^B_M} : L^B_M \Rightarrow \mathcal{MM}_{\mathcal{B}}$: The levels $L^B_{M,i}$ for all $0 < i \leq n_b$ of the inclusion chain will be produced by the pushouts of the spans $L^B_{A,i} \hookleftarrow L^B_{0,i} \hookrightarrow L^B_{B,i}$.

Figure 14 illustrates how the levels $L^A_{M,0}$ and $L^A_{M,1}$ are constructed (the relations between the levels are omitted to simplify the diagrams). The other levels, as well as the pushouts with respect to $\mathcal{MM}_{\mathcal{B}}$, are constructed analogously. $L^A_{M,0}$ and $L^A_{M,1}$ are obtained by the pushouts of the spans $L^A_{A,0} \hookleftarrow L^A_{0,0} \hookrightarrow L^A_{B,0}$ and $L^A_{A,1} \hookleftarrow L^A_{0,1} \hookrightarrow L^A_{B,1}$, respectively. The graphs $L^A_{0,1}$ and $L^A_{B,1}$ will be empty since the inclusion chain is constructed with respect to $\mathcal{MM}_{\mathcal{A}}$. This is because the elements of L_0 and L_B have only the default types in $\mathcal{MM}_{\mathcal{A}}$, hence only level 0 of these inclusion chains are none-empty such that $L_0 = L_{0,0}$ and $L_B = L_{B,0}$. The construction of the two first levels for the rules TR_A and TR_B from the running example is shown in Figure 15. Notice that d is neither identified in $L^A_{0,1}$ nor in $L^A_{B,1}$, as it only has the default EClass type w.r.t. $\mathcal{MM}_{\mathcal{A}}$, located in level 0. Then, in $L^A_{M,1}$ we only have a, which is typed by A in $MM_{A,1}$.

To summarise, the result of the amalgamation process is an amalgamated rule where each element has two types. For the running example, the result of

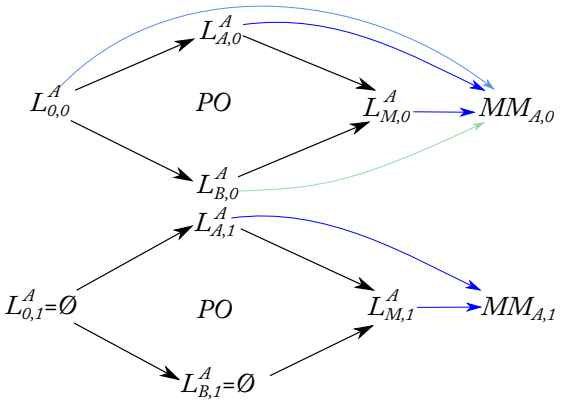


Figure 14: Levels $0, 1$ of the inclusion chain $\mathcal{L}_{\mathcal{M}}$

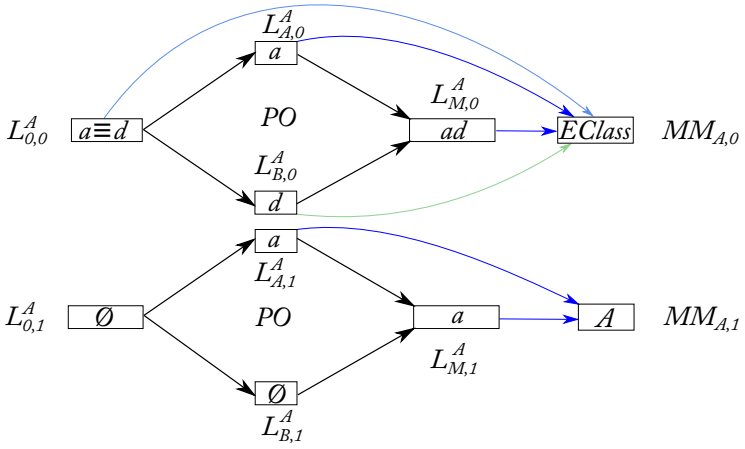


Figure 15: Levels $0, 1$ of the inclusion chain $\mathcal{L}_{\mathcal{M}}$ for the rules TR_A and TR_B

amalgamating TR_A and TR_B in Figure 11 is the rule TR_M which is depicted in Figure 16 as a co-span and in Figure 17 in the MultEcore syntax.

3.2.3. Amalgamation cases

If we inspect the constructions described in Section 3.2.2, we can observe several amalgamation cases depending on how TR_A and TR_B are related by TR_0. Table 1 shows a summary of the cases that we contemplate, which are listed below (note that one can see in the *Amalgamation* columns which elements are identified, as the names are concatenated):

Case 1 : TR_A adds, TR_B adds and $I_0 = L_0$, i.e., added elements are not identified (only a is identified with d which was already existing in L_0).

Case 2 : TR_A adds, TR_B adds and $L_0 \sqsubset I_0$, i.e. the elements newly added by each of the rules are identified between them (for example in this case, b

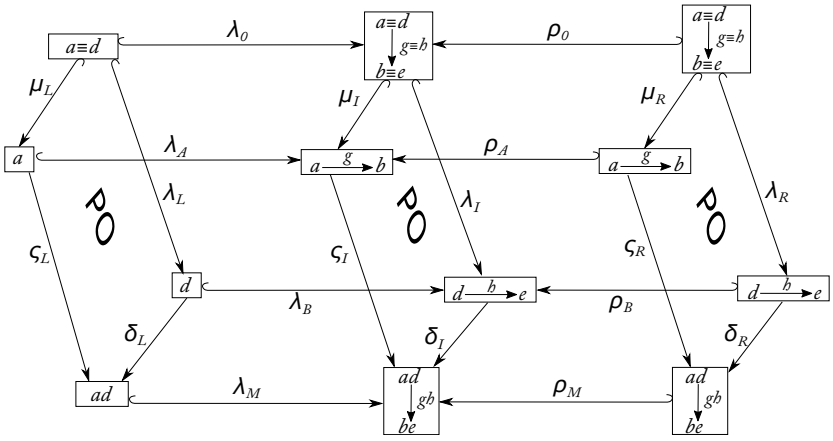


Figure 16: Amalgamation construction application of the situation depicted in Figure 17

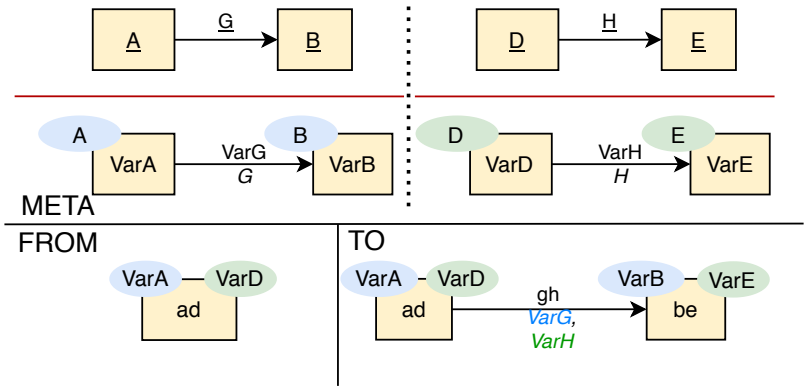
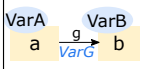
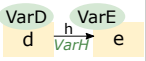
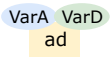
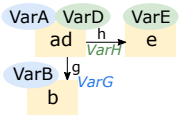
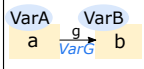
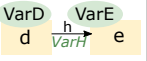
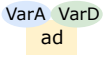
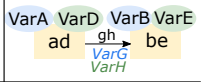
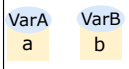
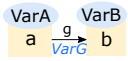
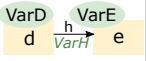
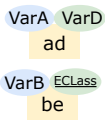
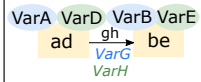
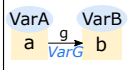
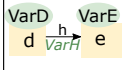
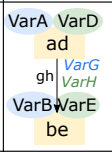
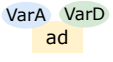
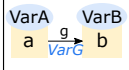
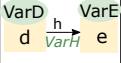
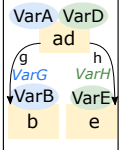
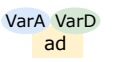
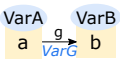
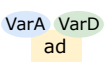
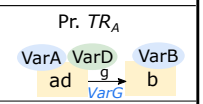
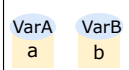


Figure 17: Amalgamated rule TR_M as result of combining TR_A and TR_B

is identified with e and g with h, which are all newly added). This case represents the example shown along Section 3.2.2

Case 3 : TR_A adds, TR_B adds, $L_0 \sqsubset I_0$ and either $(I_0 \setminus L_0) \cap L_A \neq \varnothing$ or $(I_0 \setminus L_0) \cap L_B \neq \varnothing$, i.e., newly added elements by TR_A are identified with elements which are in L_B, or vice versa. This is a special case since, for as, b is identified with e in I_0, but e does not exist in L_0. Therefore, as hinted in the constructions shown in Section 3.2.2, we need to constrain the match of L_M in the source graph S by forcing the missing type for be to be directly — i.e., not transitively — EClass. In abuse of notation, we indicate with underlined text in the type rather than in the element (as we do for constants) that the constrained typing relation must match exactly one typing relation in the target hierarchy, instead of a potential series of transitive typing relations.

Table 1: Amalgamation cases

CASE	Rule A (TR_A)		Rule B (TR_B)		Amalg. (TR_M)	
	L_A	R_A	L_B	R_B	L_M	R_M
1						
2						
3						
4						
5						
6						 Pr. TR_B 
7						

From this point, the cases which include deletion of elements might cause a general dangling arrow problem, which has to be solved. One solution is to get rid of the dangling arrows using a special graph minus operator as explained below. Alternatively, we could notify the user about the dangling arrows and ask for user intervention as it is done, for instance, in version control systems (see [45]).

Case 4 : TR_A deletes and TR_B deletes, where $L_0 = I_0$ (i.e., there are no new identifications except for the ones in L_0), $L_M = I_M$ (no additions), and $I_A \setminus R_A = I_B \setminus R_B$ (identified/same elements are deleted). Note, if the only deleted elements are those that have been identified, we have $I_0 \supseteq I_M \setminus R_M$.

Case 5 : TR_A deletes, TR_B deletes, $I_0 = L_0$, $L_M = I_M$, $I_A \setminus R_A \neq I_B \setminus R_B$ (different elements are deleted) and $R_M \supseteq I_0$ (all the identified elements are preserved).

We will now analyse other cases involving deletion which could be covered if R_M is created by pushout of $R_A \hookleftarrow R_0 \hookrightarrow R_B$. However, if we use such a mechanism to construct R_M, we would lose the effect of deletion, and certain conflicts might just disappear. For example, if TR_A deletes an element while TR_B keeps it, the element would be kept. Obviously, a potential dangling arrow problem would also disappear since a deleted node a which is identified in L_0 or I_0 would be kept if it is preserved by the rule which uses a as source or target of an arrow. Therefore we introduce two priority formulae below to prioritise the effect of one of the rules depending on the user's choice (the calculation of R_M, i.e., the square on the far right of Figure 12, would be done via the formulae below).

$$\text{Priority in } TR_A: \quad R_M = R_A \cup (R_B \underline{\setminus^*} (I_0 \cap R_B))$$
$$\text{Priority in } TR_B: \quad R_M = R_B \cup (R_A \underline{\setminus^*} (I_0 \cap R_A))$$

We define $\underline{\setminus^*}$ as a graph minus operation that removes any dangling arrow that could be left by the usual graph minus operation.

To illustrate an application of priorities, let us consider the example shown in Figure 18 where we have two rules: TR_A:= *Add and connect* and TR_B:= *Delete node*. The latter was originally conceived to be applied to the right-hand branch of the hierarchy (*specific-model-2, generic-model-2, Ecore*) in Figure 8. In the case of the *Delete node* rule, and following the same logic as explained for the *Add and connect* rule, any match in our instance of the variable d placed in the FROM block, whose type is VarD located at the second level of the META block, and which is typed by the constant D located at the first level of the META block, takes the instance to a new state where the matched element is removed. These rules are conflicting in the sense that the user has identified a with d and, while TR_A adds a new arrow g to a, TR_B deletes the element d. However, as mentioned, applying our standard pushout construction would produce a TR_M in which the affect of the deletion in R_M disappears.

Depending on which rule the user wants to prioritise, the corresponding formula needs to be applied. First, the user has to provide I_0 with the identification. In this case, I_0 only identifies a with d (a ≡ d). Such an identification

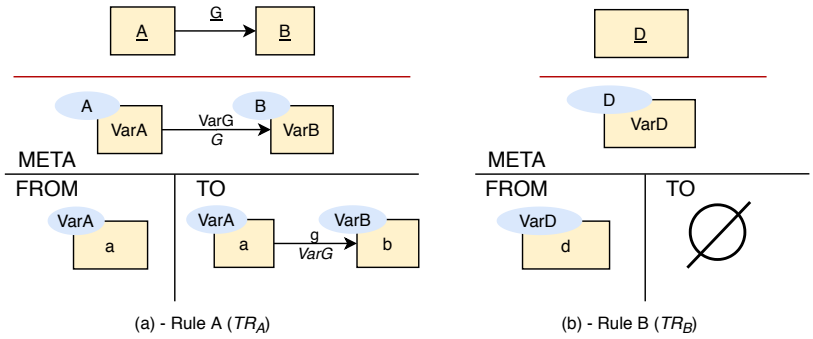


Figure 18: MCMT rules to be amalgamated: (a) Rule A affecting the left branch and (b) Rule B affecting the right branch

indicates us that a, d, or ad appearances in the formula must be treated as same element. If the prioritisation falls on TR_A, we have:

$$R_M = (a \xrightarrow{g} b) \cup (\varnothing \underline{*} (ad \cap \varnothing))$$
$$R_M = (a \xrightarrow{g} b)$$

If the prioritisation is given to TR_B the result is:

$$R_M = \varnothing \cup ((a \xrightarrow{g} b) \underline{*} (ad \cap (a \xrightarrow{g} b))$$
$$R_M = \varnothing \cup ((a \xrightarrow{g} b) \underline{*} a)$$
$$R_M = b$$

Observe how a normal minus operation would keep a dangling arrow pointing to b, while the $\underline{*}$ operation removes also the arrow. We graphically show both

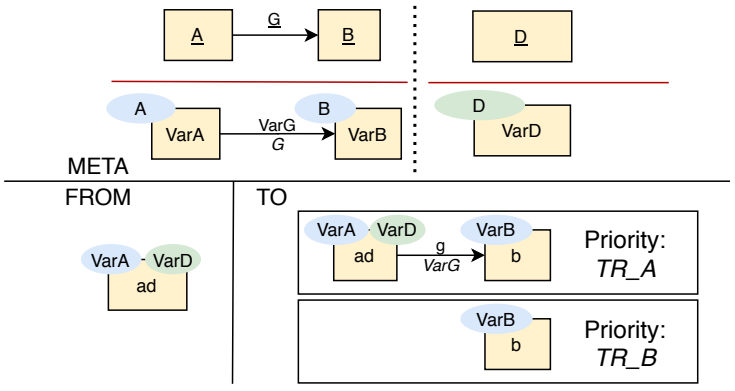


Figure 19: Amalgamated rule TR_M as result of combining TR_A and TR_B where R_M has been calculated with the priority formulae

possible results in Figure 19, where the TO block depicts the two alternatives depending on the priority.

Case 6 : One of the rules adds while the other deletes, for instance, TR_A adds something to an element while TR_B deletes that element. This is the case depicted above and shown in Figures 18 and 19 where R_M is given by prioritisation on one of the rules.

Case 7 : This case covers potential combinations of some of the cases afore discussed. There might be several additions and/or deletions at the same time and, therefore, conflicts that would require prioritisation.

3.2.4. Amalgamated rule application

The last step, once the amalgamated multilevel double-typed rule is constructed, consists of its application into the composed multilevel hierarchy. Note that the construction follows the same reasoning as for single multilevel typed rules (detailed in Section 2.2). The complete construction for the amalgamated rule application is depicted in Figure 20.

As we discussed in Section 3.2.3, the calculation of R_M might not be done by the pushout but with the priority formulae, so that we mark the right hand pushout with $*$. We have the two typing chains $\mathcal{MM}_\mathcal{A} = (\overline{MM_A}, n_a, \tau^{MM_A})$ and $\mathcal{MM}_\mathcal{B} = (\overline{MM_B}, n_b, \tau^{MM_B})$ over which the double-typed MCMT rule is defined. The multilevel double-typed rule is given by the four components (L_0, L_A, L_B and L_M for L and respectively for I and R) and their multilevel typings over the two typing chains $\mathcal{MM}_\mathcal{A}$ and $\mathcal{MM}_\mathcal{B}$ such that $\sigma^{L_A^A} : L_A \Rightarrow \mathcal{MM}_\mathcal{A}$ and $\sigma^{L_B^B} : L_B \Rightarrow \mathcal{MM}_\mathcal{B}$, $\sigma^{I_A^A} : I_A \Rightarrow \mathcal{MM}_\mathcal{A}$ and $\sigma^{I_B^B} : I_B \Rightarrow \mathcal{MM}_\mathcal{B}$ and $\sigma^{R_A^A} : R_A \Rightarrow \mathcal{MM}_\mathcal{A}$ and $\sigma^{R_B^B} : R_B \Rightarrow \mathcal{MM}_\mathcal{B}$. Then, we have $\sigma^{L_M^A} : L_M \Rightarrow \mathcal{MM}_\mathcal{A}$, $\sigma^{L_M^B} : L_M \Rightarrow \mathcal{MM}_\mathcal{B}$, $\sigma^{I_M^A} : I_M \Rightarrow \mathcal{MM}_\mathcal{A}$, $\sigma^{I_M^B} : I_M \Rightarrow \mathcal{MM}_\mathcal{B}$, $\sigma^{R_M^A} : R_M \Rightarrow \mathcal{MM}_\mathcal{A}$ and $\sigma^{R_M^B} : R_M \Rightarrow \mathcal{MM}_\mathcal{B}$.

In the multilevel typed setting all the instance graphs S, D and T are multilevel double-typed over another two typing chains $\mathcal{TG}_\mathcal{A} = (\overline{TG_A}, m_a, \tau^{TG_A})$ and $\mathcal{TG}_\mathcal{B} = (\overline{TG_B}, m_b, \tau^{TG_B})$, the instance typing chains. A **match** of the left-hand side (L_M, $\sigma^{L_M^A}, \sigma^{L_M^B}$) of the multilevel double-typed rule into a multilevel double-typed instance graph (S, $\sigma^{S^A}, \sigma^{S^B}$) is given by a graph homomorphism $\mu_M : L_M \to S$ together with the corresponding typing chain morphisms (β_A, f_A) and (β_B, f_B) where $\beta_A = \beta_{A_i} : MM_{A_i} \to \mathcal{TG}_{\mathcal{A}\, f_A(i)} \,|\, i \in [n_a]$ and $\beta_B = \beta_{B_i} : MM_{B_i} \to \mathcal{TG}_{\mathcal{B}\, f_B(i)} \,|\, i \in [n_b]$, respectively.

Furthermore, $\mu_M : L_M \to S$ has to be compatible with the multilevel typings $\sigma^{L_M^A} : L_M \Rightarrow \mathcal{MM}_\mathcal{A}$ and $\sigma^{L_M^B} : L_M \Rightarrow \mathcal{MM}_\mathcal{B}$, $\sigma^{S^A} : S \Rightarrow \mathcal{TG}_\mathcal{A}$ and $\sigma^{S^B} : S \Rightarrow \mathcal{TG}_\mathcal{B}$ and, finally, with the typing chain morphisms $(\beta_A, f_A) : \mathcal{MM}_\mathcal{A} \to \mathcal{TG}_\mathcal{A}$ and $(\beta_B, f_B) : \mathcal{MM}_\mathcal{B} \to \mathcal{TG}_\mathcal{B}$.

We construct the pushout and then the final pullback complement of the underlying graph homomorphisms in the category **Graph** as shown at the bottom of Figure 20. The type compatibility conditions for the multilevel double-typed rule as well as for the multilevel typed match should ensure that we obtain, in a canonical way, multilevel typings $\sigma^{D^A} : D \Rightarrow \mathcal{TG}_\mathcal{A}$ and $\sigma^{D^B} : D \Rightarrow \mathcal{TG}_\mathcal{B}$,

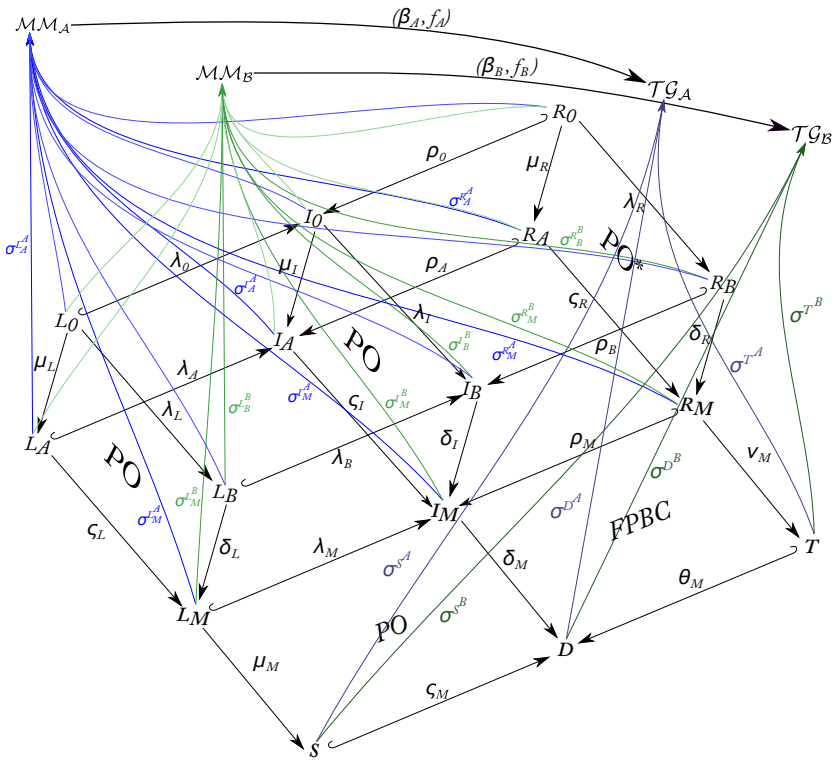


Figure 20: Amalgamated rule application construction

$\sigma^{T^A} : T \Rightarrow \mathcal{TG}_\mathcal{A}$ and $\sigma^{T^B} : T \Rightarrow \mathcal{TG}_\mathcal{B}$ of the constructed graphs such that the constructed graph homomorphisms $\varsigma_M : S \hookrightarrow D$, $\delta_M : I_M \to D$, $\theta_M : T \hookrightarrow D$ and $\nu_M : R_M \to T$ are type compatible.

4. Case study

The capability to perform composition of structure and operational semantics takes the construction of DSMLs to a next step. Modelling a system often involves the consideration of several perspectives that describe different aspects of the system. In the case study that we present in this section, the main aspect of the system consists of a DSML defined as a multilevel hierarchy for the management and distribution of process resources in a company. This is the application hierarchy of the case study, called *process management*. The *process management* hierarchy version we present in this paper is a fragment of the hierarchy presented in the MULTI 2019 workshop, as our solution [27] to the MULTI Process challenge [46, 47]. Therefore, all the modelling decisions affecting the complete hierarchy (illustrated in Appendix A) were made to fulfil the requirements of the challenge. The second DSML is described in an independent multilevel hierarchy that captures certain notions related to human beings in

general (e.g., stamina). This second hierarchy acts as the supplementary one in our case study, and it is called the *human-being* hierarchy. By applying our approach we observe that composition can be achieved in a natural and modular way. The composition of structure can be done by double typing elements, while the MCMTs can be composed by applying the constructions introduced in Section 3.2.

4.1. The process management hierarchy

This hierarchy represents the domain of process management, where the modeller is interested in a complete description of a language that includes the specification of particular occurrences (i.e., "processes" = "processes instances", "tasks" = "task occurrences") and universal kinds of occurrences ("process definitions", "task types") and relations to actor types and artefact types. Our

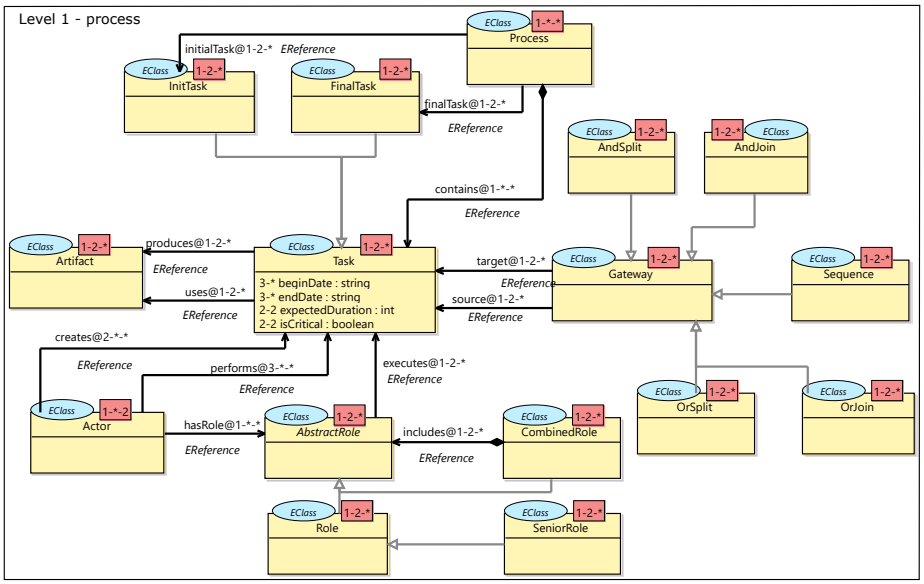


Figure 21: Process management model

original solution ([27]) presented models not only related to the general management of processes but also branches for specific processes in the domains of software engineering and insurance. For the sake of simplicity, we focus only on the software engineering branch as it suffices to illustrate our composition approach.

4.1.1. Structure of the process management hierarchy

The *process* model depicted in Figure 21 is located in the first level (we omit *Ecore*, which lives above *process* model) of the hierarchy and contains the concepts concerning universal processes. This includes *process types*, *task types*, *artefact types*, and *actors*. The composition relation named contains between

Process and Task models that a process has one or more tasks. Task has some attributes to model the duration, starting and ending day, and whether it is critical or not. Actors may have multiple roles, which is captured by the reference hasRole between Actor and AbstractRole. We use for roles the traditional object-oriented Composite pattern [48] and define AbstractRole as an abstract node (italic font in the name). A special type of role to designate a SeniorRole is also defined. Roles can have assigned kind of tasks whose instances can execute. Also, each actor can either create or perform tasks. Finally, the two references, produces and uses, from Task to Artifact, capture that tasks can both use and produce artefacts. Ordering constraints between task types are established through Gateways, which may be Sequence, OrSplit, OrJoin, AndSplit and AndJoin.

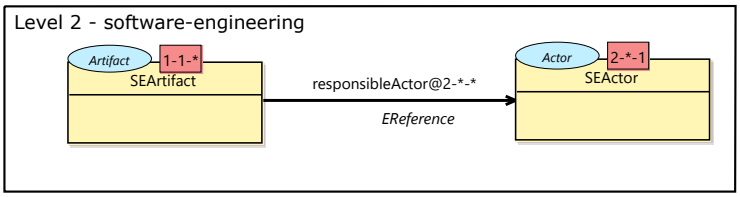


Figure 22: Software engineering process model

The model *software-engineering* (in level 2) in Figure 22 captures specialisations that affect the software engineering domain. For instance, that each soft-

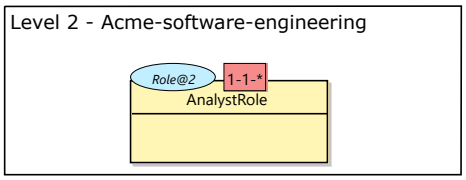


Figure 23: Acme software engineering process model excerpt

ware engineering artefact (SEArtifact node has as type Artifact from the *process* model in Figure 21) must have assigned one responsible software engineering actor.

The *Acme-software-engineering* model describes a concrete modelling language for the Acme company, and characterises how the working flow is going to be, which roles are allowed to execute certain types of tasks, which artefacts are produced, and so on. Figure 23 shows the excerpt of this model that is needed for the current case study — the entire model is depicted in Figure A.41(c). In this excerpt, we find AnalystRole class of type Role. Note that @2 is added to it type as it is located two levels above (at *process* model in Figure 21).

The lowest level of the *process management* hierarchy contains the instance model (called *Acme-configuration*) and it is shown in Figure 24. It depicts a very simple initial model with Alex as a software engineering actor (SEActor)

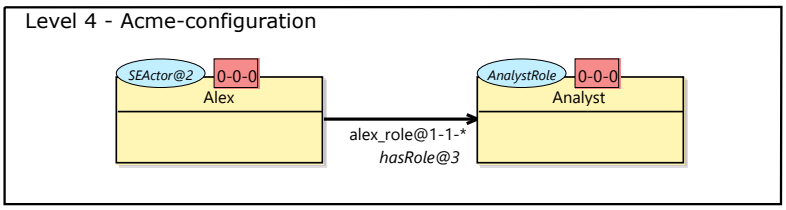


Figure 24: Acme initial configuration at the instance level

which has associated an Analyst role (of type AnalystRole).

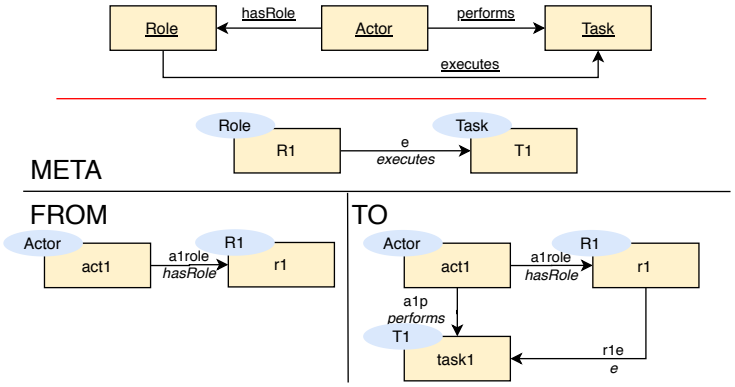


Figure 25: Rule *Create Task*: It creates a specific task associated to a concrete actor whose role allows the execution of such kind of tasks

4.1.2. MCMTs for the process management hierarchy

The dynamics of processes is modelled by MCMTs, which describe the different actions that may occur in the system. We show here three of these rules for the *process management* hierarchy that illustrate their use, and will serve us to manifest their combination with rules in the second hierarchy.

The first rule, called *Create Task*, is shown in Figure 25. Given an actor act1 with a role r1 of some type R1 via a1role, the rule assigns a new task of the right type to it. The role specified in the level 2 of the META block will constrain the task that such role can execute. In addition, the model at the higher level will similarly constrain the type of task that the actor can perform and its role execute.

The second rule, named *Produce Artefact*, is depicted in Figure 26. If an actor act1 and a task task1 he is performing (indicated by the a1p reference) are found, the rule creates an artefact ar1 related both to the actor act1 via r (typed by responsibleActor) and to the task task1 via t1pr.

The third rule that applies to the process management multilevel hierarchy, named *Delete task* and illustrated in Figure 27, is meant to delete a task that an actor is performing. Recall that rule levels are not expected to match consecutive

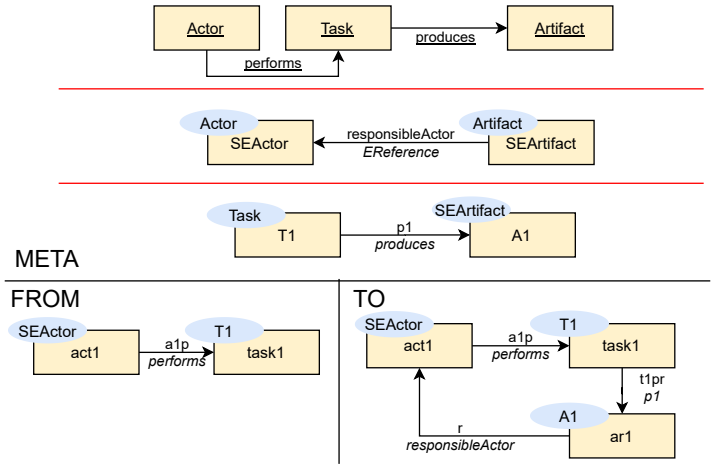


Figure 26: Rule *Produce Artefact*: It creates an artefact related to the task that produces it and the actor responsible for it

levels in the hierarchy on which they are defined. In this case, the META model would match to elements located at level 1 of the hierarchy (Figure 21), while the FROM and TO parts would match at the instance level placed at level 4 (Figure 24). This flexibility is specified in Condition 1.

4.2. The human-being hierarchy

In the *human-being* hierarchy we tackle different aspects inherently related to the human factor of the system.

4.2.1. Structure of the human-being hierarchy

This multilevel hierarchy is depicted in Figure 28. The model represented in Figure 28(a) captures very general human being notions, such as that a human (Human node) can do (does relation) multiple activities (Activity node). Furthermore, a human has a stamina level which is represented as an Integer (int), and an activity can have an impact on a human's stamina. These two characteristics are expressed via attributes in the respective nodes.

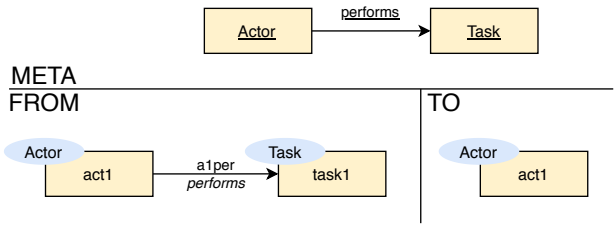


Figure 27: Rule *Delete Task*: It deletes a task an actor is performing

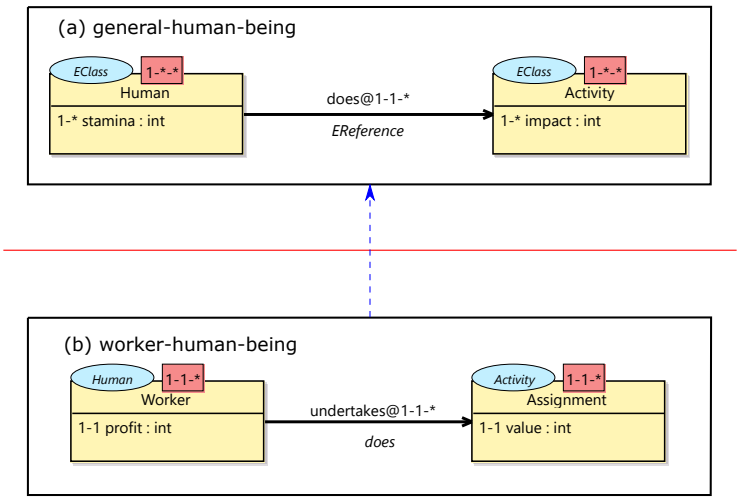


Figure 28: Human-being multilevel hierarchy

To give an example of refinement, we define in Figure 28(b) a model that captures concepts for the domain of working human beings. Note that we could add other models in here at the same level to capture other areas, such as students, retired people, etc. Worker, undertakes and Assignment have, as types, Human, does and Activity, respectively. Additionally in this level, two more attributes are added that only concern the worker domain. The profit attribute (defined in Worker) can be understood as the income that a worker obtains. And value, specified in the Assignment node, is the benefit that completing the assignment provides.

4.2.2. MCMTs for the human-being hierarchy

As we did for the process management hierarchy (Section 4.1.2), behaviour here is also described using MCMTs. We provide two MCMT rules for this hierarchy.

The first rule is called *Undertake activity* and it is shown in Figure 29. It connects a worker work1 and an assignment as1. Attributes are also modified in this rule. In the FROM block, s, p, i and v would capture values in the model for the stamina and profit for work1 and the impact and value for as1, respectively, during the matching process. In the TO block, apart from connecting them via u (typed by undertakes) reference, the attributes on the worker are modified: stamina in work1 gets decreased by the amount that was matched to the impact from the assignment as1 but the profit on the worker work1 gets increased by the amount specified in the value attribute in the assignment as1. Intuitively, a worker that is undertaking an activity gets income at the cost of getting more tired.

A second rule, named Finish Activity, is illustrated in Figure 30. Unlike the previous rule which is defined in the domain of worker human beings, this one

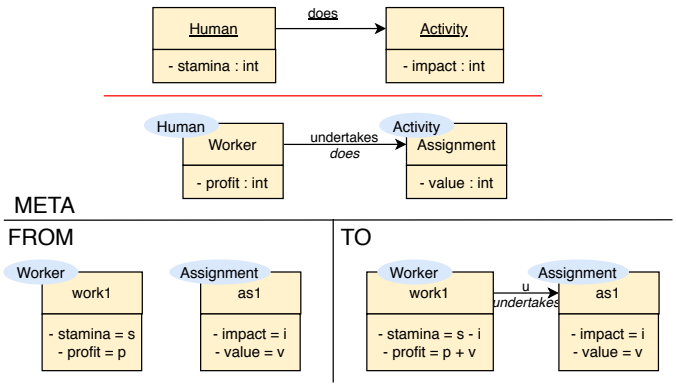


Figure 29: Rule *Undertake activity*: It connects a worker with the assignment being performed and updates its attributes

applies to human beings in general. The application of this rule finds a match in the model where a human human1 connected to an activity act1 via d and removes such a reference.

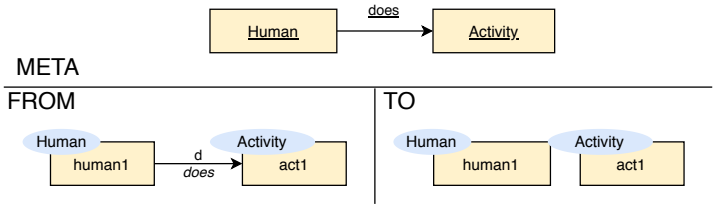


Figure 30: Rule *Finish Activity*: It removes the link between a human being and the activity he was performing

4.3. Multilevel hierarchies combination

A modeller working on a concrete design of the processes of the Acme company (specific actors, tasks, artefacts, etc.) might find useful to complement that given scenario with additional aspects, such as those described in the *human-being* multilevel hierarchy (Figure 28). Through our approach one can put together different perspectives, while there still exists a separation (via typing chains) that can be analysed either together or separately.

For instance, observe the model *Acme-configuration-composed* depicted in Figure 31 where we incorporate the *human-being* multilevel hierarchy (Figure 28) as a supplementary typing chain to reason about some elements defined on it. We can, for example, give to Alex the new type Worker and keep the SEActor type. Analogously, we can instantiate the attribute stamina, which comes from the *worker-human-being* model (Figure 28(b)), with the value 3.

To give a full perspective of how the two hierarchies are put together and how elements at instance level can make use of them, we provide selected parts

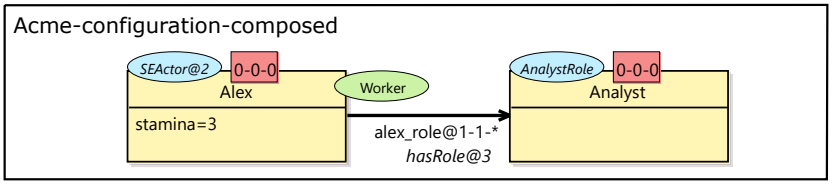


Figure 31: Instance model of Acme software engineering company including *human-being* hierarchy

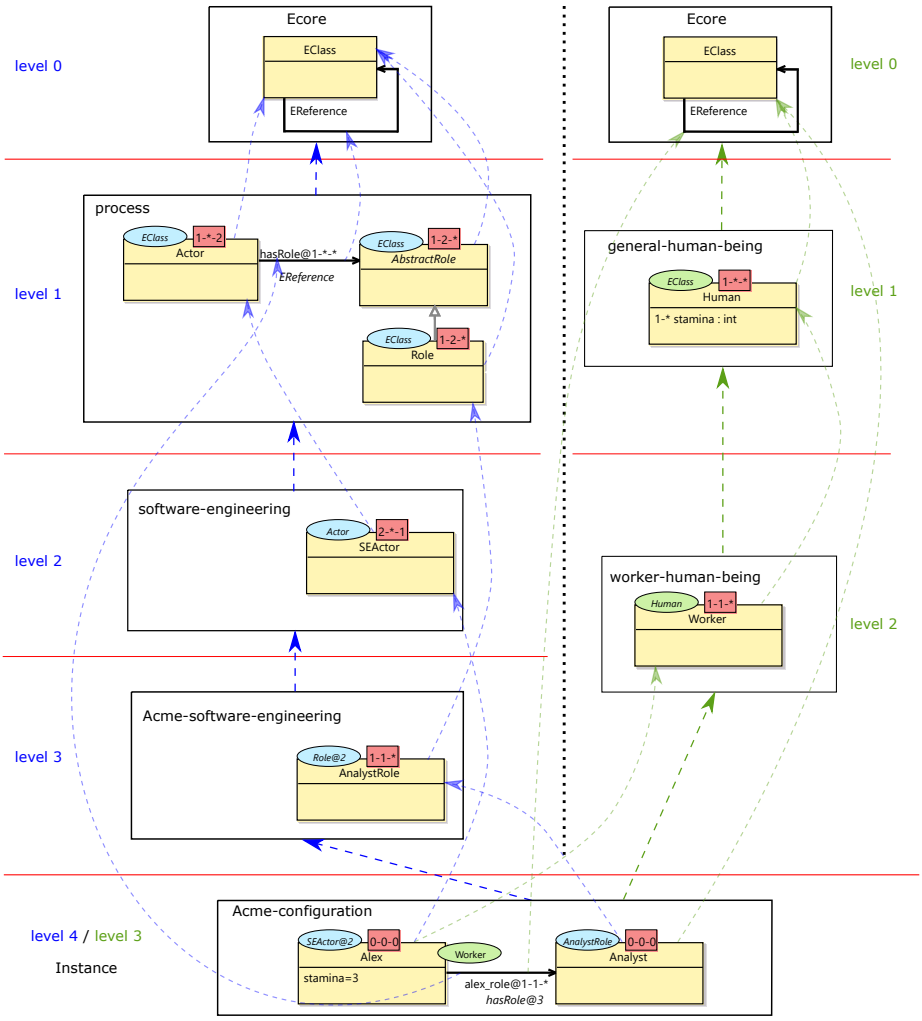


Figure 32: Selected parts of *process* and *human-being* hierarchies creating a composed multi-level hierarchy

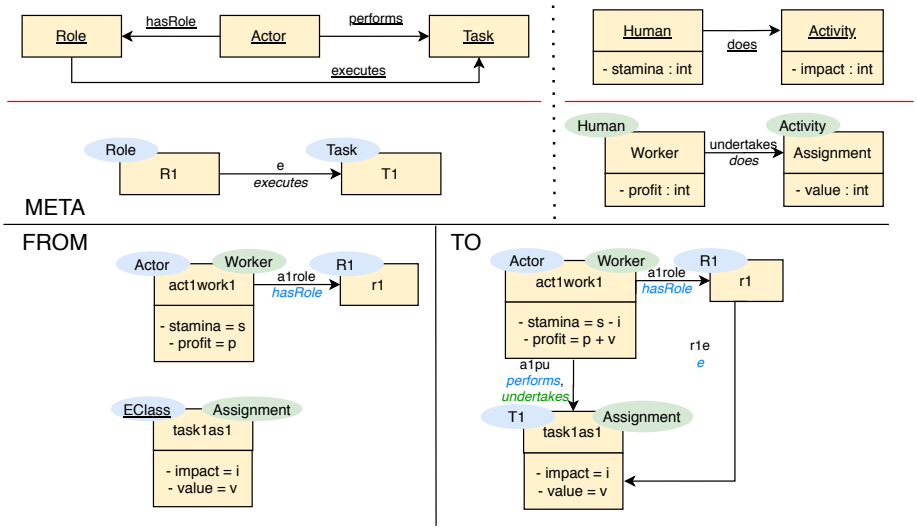


Figure 33: First rule amalgamation: It combines *Create Task* rule from the *process* hierarchy with *Undertake Activity* rule from the *human-being* hierarchy

of each of the models and illustrate them in Figure 32, where one can observe the typing chains for each hierarchy. Note that the model shown in Figure 31 is located at level 4 / level 3 - Instance in Figure 32. Each of the types belonging to each of the hierarchies can be precisely spotted in its corresponding typing chain up to the topmost model. Firstly, Alex is typed by SEActor. Note that the @2 means that SEActor is located at level: Alex's level (level 4) minus 2, i.e., at level 2 — in the *software-engineering* model. Then SEActor's type is Actor located at level 1 which finally leads us to EClass defined at level 0. Secondly, Alex's second type is Worker, which is located at level 2 (*worker-human-being* model). Worker's type is Human placed one level above (*general-human-being* model) and, ultimately, Human's type is EClass. For each of the elements present in any of the models, one must always be able to follow the typing chains up to the topmost model located at level 0. The dashed semi-transparent lines in Figure 32 represent the typing chains of each element.

4.4. MCMTs amalgamation

We show in this section, to demonstrate the application of the constructions detailed in Section 3.2, three amalgamation cases, each of them combining one rule from each hierarchy.

The first amalgamated rule shown in Figure 33 is given by the combination of the *Create Task* (Figure 25) rule from the *process management* hierarchy and the *Undertake Activity* (Figure 29) rule from the *human-being* hierarchy. We identify in the META block both multilevel hierarchies (note they are separated by a vertical dotted line) involved in the two typing chains present in the FROM

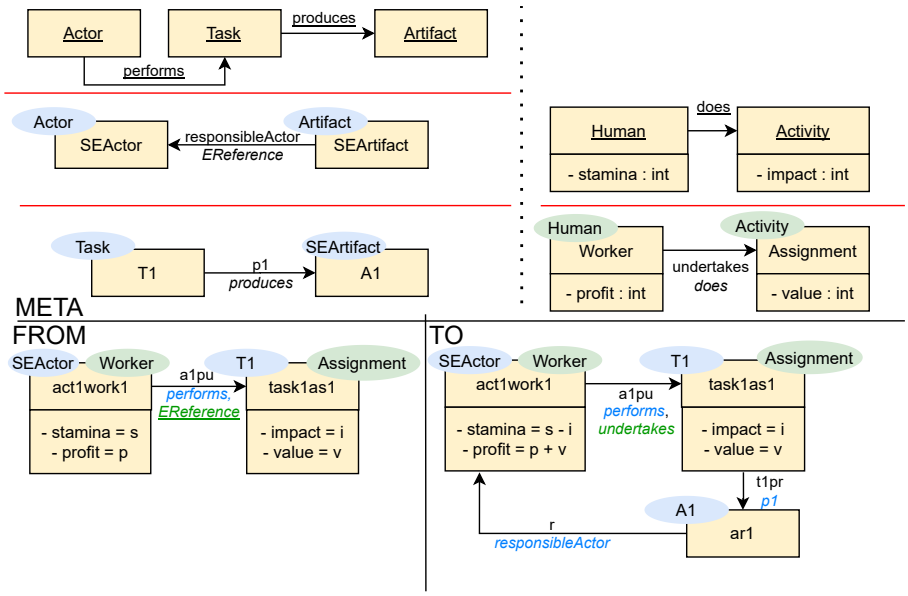


Figure 34: Second rule amalgamation: It combines *Produce Artefact* rule from the *process* hierarchy with *Undertake Activity* rule from the *human-being* hierarchy

and TO blocks, product of the amalgamation process. The complete amalgamated rule is automatically obtained by applying the construction shown in Figure 12, once I_0 has been provided by the user. This rule intuitively assigns to an actor/worker (act1work1) a task/assignment (task1as1) through a1pu, for the first hierarchy, and undertakes, for the second one. As clarified in *Case 3* of Section 3.2.3, act1work1's type from the *process* hierarchy is constrained to be EClass. The rule also connects r1 to task1as1 via r1e. Notice how r1e link is not involved with the *human-being* hierarchy, which makes sense since roles from the *process* hierarchy are not identified with anything into the *human-being* hierarchy. Finally, it also applies the attribute manipulation such as decreasing the stamina and increasing the profit of act1work1. This rule is identified by case number 3 in Table 1.

The second amalgamated rule displayed at Figure 34 is constructed by combining the *Produce Artefact* rule (Figure 26) from the *process management* hierarchy and again the *Undertake Activity* rule from the *human-being* hierarchy.

In this case, we illustrate this rule as it presents a peculiarity. As one can observe in Figure 34, there exists a mismatch between the number of levels in the two hierarchies. While the first hierarchy on the rule (located in the left-hand side of the dotted line in the META block) specifies three META levels, the second hierarchy or at the right-hand side only contains two levels. However, this is not a problem since either of the typing chains do not see themselves affected by the other, and it is perfectly fine to find such kind of situations. The application of this rule creates an artefact ar1 (which is not related to the

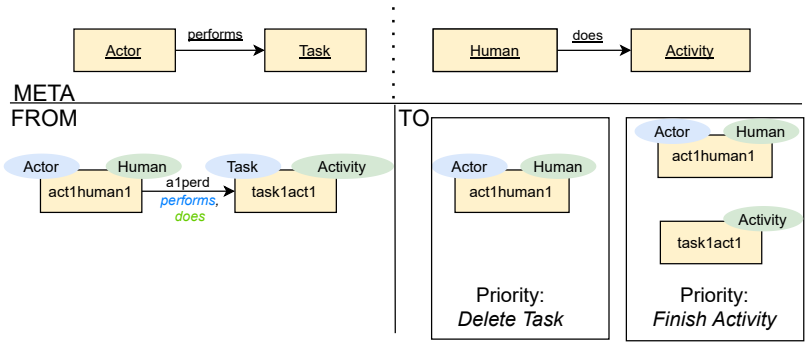


Figure 35: Third rule amalgamation: It combines *Delete Task* rule from the *process* hierarchy with *Finish Activity* rule from the *human-being* hierarchy. The result depends on which rule gets prioritised

human-being hierarchy) related to act1work1 through r and to task1as1 via t1pr. Again, act1work1 also gets updated stamina and profit. This rule construction is covered in case number 3 in Table 1 (notice the EReference second type of a1p in the FROM block).

The last amalgamation example we have obtained, is given by the combination of *Delete Task* rule (Figure 27) from the *process management* hierarchy and *Finish Activity* rule (Figure 30) from the *human-being* hierarchy. We illustrate in this case an example where prioritisation must be given to one of the rules in order the get R_M (TO block). The two results depicted in the TO part are calculated by applying the formulae given in Section 3.2.3. This example corresponds to case number 6 in Table 1, as one rule is keeping the node task1act1 while the other is removing it.

4.5. Amalgamation in MultEcore

In MultEcore, we have developed a guided-procedure that the modeller follows in order to get the set of amalgamated rules. To facilitate the explanation,

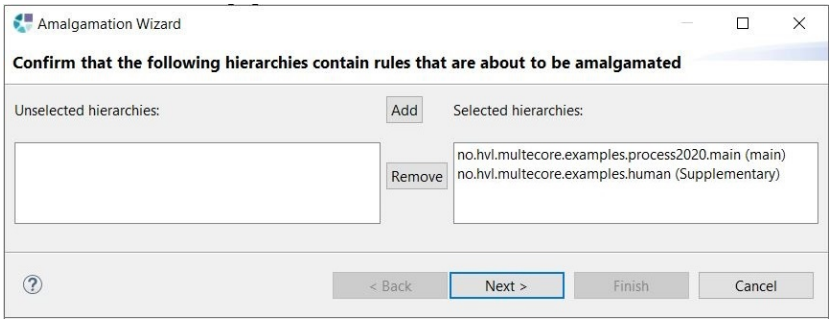


Figure 36: First step of amalgamation wizard: Selection of multilevel hierarchies to be combined

we describe each step and add a corresponding figure of how it is displayed to the user in the MultEcore wizard. We use the three amalgamated rules shown in Figures 33, 34 and 35 for illustrative purposes and to demonstrate that the produced amalgamated MCMT rules are sound with the expected results. The amalgamation process is semi-automatic, and defined by the following steps:

1. The modeller decides which multilevel hierarchies are going to be combined. These, together with their corresponding set of MCMT rules will be loaded into the wizard. This is shown in Figure 36 where both multilevel hierarchy projects have been selected. It is important to mention that one must select at least two of the available hierarchies in order to advance to the second step.
2. In this step, the user has to pick the MCMT rules that are going to be amalgamated. For instance, if we are combining two hierarchies, the modeller has to specify pairs of MCMTs that are to be amalgamated. We show in Figure 37 and describe below the four sub-steps that are involved at this stage of the wizard:
   - **Figure 37.1**. In this part of the dialog the user sees all the multilevel hierarchies that have been selected. In this example we have: ... process2020main and ... human.
   - **Figure 37.2**. In this box the user automatically sees the available MCMT rules that belong to the selected hierarchy in Figure 37.1. Selecting one of them and pressing Add Rule (right side of the Figure) adds the selected MCMT rule to the third box (Figure 37.3). Note that only one rule can be added per hierarchy and, for instance, adding two rules from the same hierarchy is not a valid situation. Once a rule has been added to Figure 37.3, it is removed from the box in Figure 37.2 until it has been resolved, i.e., combined with another rule.

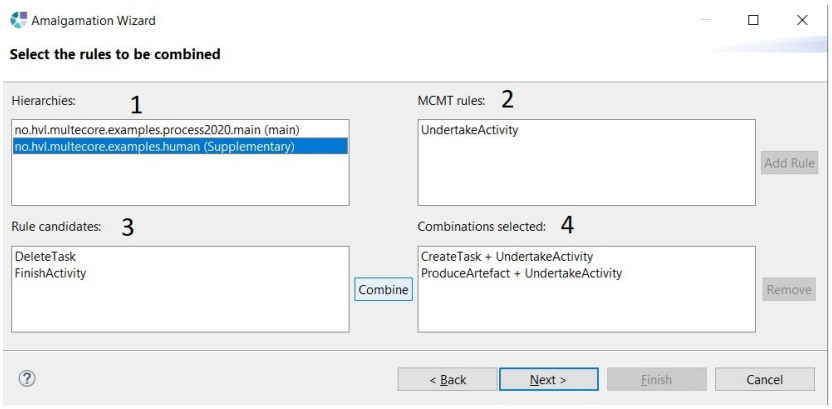


Figure 37: Second step of amalgamation wizard: Selection of MCMT rules combinations.

- **Figure 37.3**. This box shows the potential MCMT rules that are a priori candidates to be combined. Note that FinishActivity and DeleteTask are currently shown in there, and pressing Combine will save this combination together with the already decided ones.
- **Figure 37.4**. This last box shows the combinations that have been stored for next steps of the wizard. Currently, two combinations have already been decided: CreateTask + UndertakeActivity and ProduceArtefact + UndertakeActivity. Combinations can be discarded by selecting one and clicking on Remove (bottom right of Figure 37). For the next step of the wizard we assume that the candidate combination in Figure 37.3 is finally combined.

3. For each tuple of assigned MCMTs, one needs to give the identification of the elements. As mentioned earlier, an essential step to achieve amalgamation (or, in general, composition) is the identification process where the elements that correspond to each other have to be identified (I_0). Several approaches in the literature use a so-called kernel rule to express correspondences between two or more rules [40, 41, 49, 42]. Thus, a mandatory step within this process for the user is to provide the correspondences between elements in the rules which are to be amalgamated. We show in Figure 38 the ongoing process of the identification of each node/edge for each rule combination and describe each part:
   - **Figure 38.1**. In this box the user can see the combinations selected in the previous step. Note that we were actually showing two combinations in Figure 37.4 and the last one (DeleteTask and FinishActivity in Figure 37.3) we assume it has been combined at this point. Clicking one of these combinations and pressing Select, reveals each MCMT rules involved in such a combination in the Figure 38.2 box.

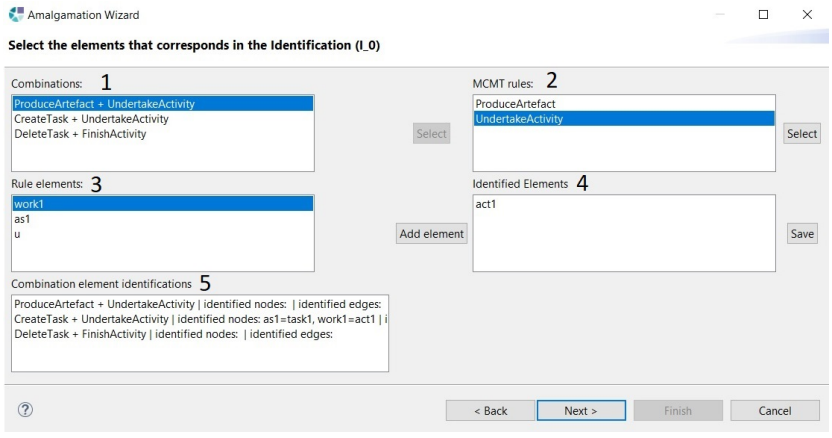


Figure 38: Third step of the amalgamation wizard: Identification of elements in each rule combination.

The three available combinations are: ProduceArtefact + UndertakeActivity (selected), CreateTask + UndertakeActivity and DeleteTask + FinishActivity.

- **Figure 38.2**. In here the MCMT rules of the combinations are shown. The user can click on one of these rules and press Select which breaks down all of the available elements of the selected rule in the **Figure 38.3** box. In the example UndertakeActivity is selected.
- **Figure 38.3**. The individual elements that belong to the selected rule are shown in this part. To choose one to identify it with another corresponding element, click on it and press Add element. In the example, work1 (selected), as1 and u are the candidates elements, and, for example, pressing Add element will move work1 to the Figure 38.4 list. Note that, similarly as in the second step with the rules to be combined, only one element per rule can be selected for one combination, and adding it as a candidate for the identification temporarily removes it from the list in Figure 38.3. In this case, the combination taking place considers one element from UndertakeActivity (work1) and one from ProduceArtefact (act1 which already added in **Figure 38.4**).
- **Figure 38.4**. The identified candidate elements are shown in this box. Currently, only act1 is on the list. Adding work1 from the previous sub step will complete this identification list and will allow the Save button to be pushed, which adds the identification to the pool.
- **Figure 38.5**. This last box simply informs of the current status of the saved identified elements in each amalgamation.

4. Finally, once all the correspondences are established, the modeller gets a summary and is notified if *conflicts* have been detected. As discussed in Section 3.2.2, a conflict may appear, for instance, when an identified node is removed in one of the selected MCMTs, but kept in the other. Our way to resolve conflicts is by granting prioritisation to one of the rules. We show in Figure 39 the last step of the amalgamation wizard where the summary of the MCMT rules that are going to be amalgamated is provided. This step is divided into three categories:
   - **Figure 39.1**. Here the conflicting amalgamated MCMT rules are listed. In this case, there is only one conflicting situation, DeleteTask + FinishActivity. Picking it and pressing Select leads to the second sub-step.
   - **Figure 39.2**. The user can select in this box the rule that should get prioritised. In this example we have chosen DeleteTask.
   - **Figure 39.3**. This last part summarises the amalgamation cases that are going to be produced. Note that we are showing here the DeleteTask... situation to display how would it look like once the

MCMT rule that is going to get prioritised is selected, i.e., by pushing Select in Figure 39.2.

Once the Finish button is selected the engine computes the amalgamated MCMT rules based on the identifications provided and the prioritisations given.

4.6. Textual DSML for MCMTs

MCMT rules in MultEcore are specified using a textual editor where the MCMTs DSML [24, 25] has been built using Xtext [50]. This DSML provides the specification of modules containing a collection of MCMT rules defined independently of the hierarchy. The combined rules produced by the amalgamation engine have the same format than the MCMT rules that the user could manually write. Thus, the amalgamation results can be directly translated into an MCMT file. An example of the results that are obtained is shown in Figure 40.

For the sake of simplicity, we only show the textual representation of the third amalgamated rule DeleteTaskFinishActivity (with priority on *Delete Task*) which was graphically displayed in Figure 35. The other two amalgamated rules are shown in Appendix B (Figures B.42, B.43). In Figure 40, we distinguish three main blocks, the meta, the from and the to (lines 2, 22 and 29, respectively). In the from and to blocks we can define patterns according to the elements previously declared in the meta part. The meta block must contain a valid, non-empty pattern, but the from and to blocks may be empty. Within the textsfmeta we can define constant and variable elements, but we can only define variables in the from and to parts. They contain the same information that the corresponding blocks shown in the graphical rule.

Constant nodes are defined, for instance, as in line number 4 Actor: $process[1]!Actor where Actor is the name of the constant node, $ is used to denote

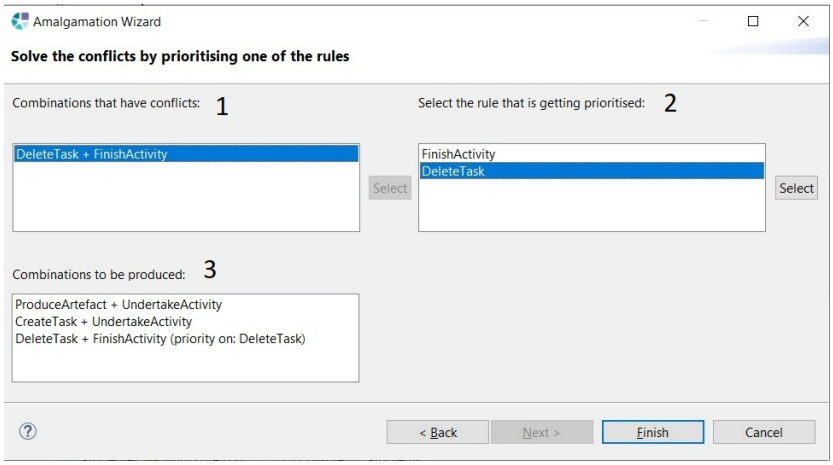


Figure 39: Fourth step of the amalgamation wizard: Conflicts resolution and summary of the MCMT rules that are going to be amalgamated.

```
1  rule DeleteTaskFinishActivity{
2          meta{
3                  //Nodes level 1 - Process
4                  Actor: $process[1]!Actor
5                  Task: $process[1]!Task
6
7                  //Nodes level 1 - Human
8                  Human: $human[1]!Human
9                  Activity: $human[1]!Activity
10
11                 //Edges level 1 - Process
12                 performs: $process[1]!Actor.performs
13
14                 //Edges level 1 - Human
15                 does: $human[1]!Human.does
16
17                 //Source.edge = Target
18                 [Actor.performs = Task]
19
20                 [Human.does = Activity]
21         }
22         from {
23                 act1human1: Actor, Human
24                 task1act1: Task, Activity
25                 a1perd: performs, does
26
27                 [act1human1.a1perd = task1act1]
28         }
29         to {
30                 act1human1: Actor, Human
31         }
32 }
```

Figure 40: Computed DeleteTaskFinishActivity MCMT rule. It corresponds to the graphical MCMT rule depicted in Figure 35 with priority on *Delete Task*

that is a constant, process is an alias of the rule it belongs to (either process or human) and [1] represents that it is located at level 1 of the meta block. Constant edges, such as the one defined in line number 12, are given by its name (performs) and ends with the form source.edge (Actor.performs). In this rule there are not variables defined in the meta block, but they are very similar with the exception that the $ is not written, and the nodes end with its type name. Also, attributes can be declared below each node specifying its type. We refer the reader to Figures B.42 and B.43 for some examples of variables and attributes. At the end of the meta block, we define the assignment expressions that are used to specify the structural relationships between the declared nodes by means of the declared edges. An example is given in line 18 [Actor.performs = Task], where Actor and Task are the source and target of the edge, performs. In the example, the from block of the rule defines a pattern consisting of three variables and one assignment expression, while its to block comprises just one

variable declaration. The from and to blocks follow the same structure. Nodes and edges in these levels are defined as shown in lines 24 and 25, respectively, where, for example, task1act1 has two types, Task from the main process hierarchy and Activity from the supplementary human one. Similarly as for the meta block, edges have to be specified within assignment expressions that link them with its respective sources and targets (line 27).

5. Related work

We first discuss approaches within the context of traditional MDSE and the Language Product Lines Engineering field that propose techniques to achieve composition.

Melange [51] is a tool for the construction of DSLs that supports modular language design and language modules composition. The dynamic semantics is defined operationally as aspects in the Kermeta meta-language [52]. Operational semantics of a DSL involves the use of an action language to define methods that are statically introduced in the concepts of the DSL abstract syntax. In our approach, we define the semantics separately, by means of MCMTs, avoiding the need to change the abstract syntax (for us, the multilevel hierarchy) of the DSML. Authors present in [53] an approach for building product lines of metamodels. The key point of these approaches is that a *transformation product line* is defined that becomes applicable for all metamodels in the set providing reusability and flexibility. Even though such approaches typically require specifying a binding between the transformation interface and the metamodel, the range of applicability is much wider than approaches where the transformation can be reused on a closed fixed metamodel set [54]. The approach in [54] is based on *featured model transformations (FMTs)* that can be seen as a kind of metamodel that integrates the variability of a whole family of metamodels which still provide a high degree of reusability. In our approach we go one step further as we do not only consider the reusability of the transformation rules within the same family, but also the incorporation of orthogonal languages.

GeKo [55] is a generic, extensible model weaver that can compose any models that conform to a common metamodel. To operate, it takes as parameters a base model, a pointcut model (the parametric pattern) and an advice model. The tool replaces all instances of the pointcut model that are found in the model with the advice model. While this approach focuses on the composition at the model (instance) level, we discuss in this work the composition of language descriptions via multilevel modelling hierarchies. Furthermore, GeKo operates only on the structure, while our approach also provides support for the amalgamation of dynamic semantics specified by means of MCMTs. MATA [56] is very similar to GeKo but it is founded on graph transformations to do composition of structure of models conforming to a common metamodel.

The work presented in [57] served us as inspiration to develop our approach. In their work, the authors formally define how composition of structure and amalgamation of semantic specifications can be achieved between a functional DSL and several parametric non-functional ones. While they establish a weaving

process to construct the combined, final products (both structure- and semantic-wise), we try to be as minimally invasive as possible by incorporating the (supplementary) typing chains which can be later removed in a flexible way. Thus, as mentioned along this article, our structure combination process tends to be *virtual* rather than *physical* in the sense that we do not produce a new combined language, but incorporate/remove the new features we are interested in.

In the context of amalgamation of graph transformations, the authors implement rule amalgamation based on nested graph predicates in GROOVE [58]. In there, a single structure holds the different rules, where pattern rules can indicate the variations of the overall pattern structure. AToM^3 supports the amalgamation of rules to describe the explicit definition of interaction schemes in different rule editors [43]. The authors of the GReAT tool [59], define the concept of *Group*, so they can operate and apply delete, move or copy operations to each of the elements within the group, in the context of a transformation rule. In our approach we explore an alternative method to achieve amalgamation based on multiple typing.

6. Conclusions and future work

In this paper we have described an alternative method to achieve composition of structure and semantics of model descriptions. While some standard approaches might achieve composition, e.g., by implementing a merge operator, we take advantage of the notion of *application* and *supplementary* hierarchies to provide elements with more than one aspect by multiple typing them. Our formalisation based on category theory and graph transformations allows us to achieve such aspect-orientation flavor by incorporating additional typing chains. We have formally demonstrated how amalgamated MCMT rules can be generated by computing their components (namely, L_M, I_M and R_M) via pushouts $L_A +_{L_0} L_B$, $I_A +_{I_0} I_B$ and $R_A +_{R_0} R_B$. We differentiate between rules that are conflict free and those whose amalgamation would lead to conflicts. For the latter, we define an alternative formulation to compute R_M, based on which rule gets prioritised. Finally, we have illustrated and applied the constructions to a case study where two independent multilevel hierarchies are combined and their rules are amalgamated. Note that we rely on the user to provide the modulo components that make it possible to calculate the resulting constructions. We are investigating how to make this process (semi-)automatic by analysing how elements at the instance level are related and multiple typed to suggest and automatically compute amalgamated rules.

The MultEcore framework is currently supporting the amalgamation process described in Sections 4.5 and 4.6. We plan to incorporate the execution of composed hierarchies with their amalgamated MCMT rules into our MultEcore-Maude infrastructure that allows to handle simulation/execution [29]. Also, we plan to extend our case studies with other examples that allow us to evaluate all cases depicted in Table 1.

References

[1] C. Atkinson, T. Kühne, Processes and products in a multi-level metamodeling architecture, International Journal of Software Engineering and Knowledge Engineering 11 (06) (2001) 761–783.

[2] S. Zschaler, P. Sánchez, J. P. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. Araújo, U. Kulesza, VML* - A Family of Languages for Variability Management in Software Product Lines, in: Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers, 2009, pp. 82–102. doi:10.1007/978-3-642-12107-4_7.

[3] J. de Lara, E. Guerra, Deep meta-modelling with MetaDepth, in: Objects, Models, Components, Patterns, Vol. 6141, 2010, pp. 1–20. doi:10.1007/978-3-642-13953-6_1.

[4] C. Atkinson, R. Gerbig, Flexible deep modeling with Melanee, in: S. Betz, U. Reimer (Eds.), Modellierung 2016, Vol. 255 of LNI, Gesellschaft für Informatik, Bonn, 2016, pp. 117–122.

[5] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, H. Ergin, AToMPM: A web-based modeling environment, in: MODELS-JP 2013, Vol. 1115 of CEUR Workshop Proceedings, 2013, pp. 21–25.

[6] S. Van Mierlo, B. Barroca, H. Vangheluwe, E. Syriani, T. Kühne, Multi-level modelling in the Modelverse, in: MULTI@ MoDELS, Vol. 1286 of CEUR Workshop Proceedings, 2014, pp. 83–92.

[7] UML, http://www.uml.org/.

[8] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: eclipse modeling framework, Pearson Education, 2008.

[9] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernández, B. Nordmoen, M. Fritzsche, Where does model-driven engineering help? Experiences from three industrial cases, Software & Systems Modeling 12 (3) (2013) 619–639.

[10] J. Whittle, J. Hutchinson, M. Rouncefield, The state of practice in model-driven engineering, IEEE software 31 (3) (2014) 79–85.

[11] J. D. Lara, E. Guerra, J. S. Cuadrado, When and how to use multilevel modelling, ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (2) (2014) 12.

[12] C. Atkinson, T. Kühne, Reducing accidental complexity in domain models, Software & Systems Modeling 7 (3) (2008) 345–359.

[13] C. Atkinson, T. Kühne, In defence of deep modelling, Inf. Softw. Technol. 64 (2015) 36–51. doi:10.1016/j.infsof.2015.03.010.

[14] C. Atkinson, R. Gerbig, T. Kühne, Comparing multi-level modeling approaches, in: Proceedings of the Workshop on Multi-Level Modelling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), Valencia, Spain, September 28, 2014, 2014, pp. 53–61.

[15] C. Atkinson, T. Kühne, On evaluating multi-level modeling, in: Proceedings of MULTI @ MODELS, 2017, pp. 274–277.

[16] F. Macías, U. Wolter, A. Rutle, F. Durán, R. Rodriguez-Echeverria, Multilevel Coupled Model Transformations for Precise and Reusable Definition of Model Behaviour, Journal of Logical and Algebraic Methods in Programming 106 (2019) 167–195. doi:10.1016/j.jlamp.2018.12.005.

[17] J. de Lara, E. Guerra, Generic Meta-modelling with Concepts, Templates and Mixin Layers, in: Model Driven Engineering Languages and Systems - 13th International Conference, MODELS, 2010, pp. 16–30. doi:10.1007/978-3-642-16145-2_2.

[18] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, B. Baudry, Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review, Computer Languages, Systems & Structures 46 (2016) 206–235. doi:10.1016/j.cl.2016.09.004.

[19] J. Kienzle, G. Mussbacher, B. Combemale, J. Deantoni, A unifying framework for homogeneous model composition, Software & Systems Modeling 18 (5) (2019) 3005–3023.

[20] Arne Lange and Colin Atkinson, Multi-level modeling with MELANEE, in: Proceedings of MULTI @ MODELS, 2018, pp. 653–662.

[21] J. de Lara, E. Guerra, Refactoring Multi-Level Models, ACM Trans. Softw. Eng. Methodol. 27 (4) (2018) 17:1–17:56. doi:10.1145/3280985.

[22] C. Atkinson, T. Kühne, J. de Lara, Editorial to the theme issue on multi-level modeling, Software and Systems Modeling 17 (1) (2018) 163–165. doi:10.1007/s10270-016-0565-6.

[23] S. P. Jacome-Guerrero, J. de Lara, TOTEM: Reconciling multi-level modelling with standard two-level modelling, Computer Standards and interfaces In press.

[24] F. Macías, A. Rutle, V. Stolz, R. Rodriguez-Echeverria, U. Wolter, An Approach to Flexible Multilevel Modelling, Enterprise Modelling and Information Systems Architectures 13 (2018) 10:1–10:35. doi:https://doi.org/10.18417/emisa.13.10.

[25] F. Macías, Multilevel modelling and domain-specific languages, PhD thesis, Western Norway University of Applied Sciences and University of Oslo (2019).

[26] F. Macías, A. Rutle, V. Stolz, Multilevel Modelling with MultEcore: A Contribution to the MULTI 2017 Challenge, in: Proceedings of MULTI @ MODELS, 2017, pp. 269–273.

[27] A. Rodríguez, F. Macías, Multilevel Modelling with MultEcore: A Contribution to the MULTI Process Challenge, in: Proceedings of MULTI @ MODELS, 2019, pp. 152–163. doi:10.1109/MODELS-C.2019.00026.

[28] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All about Maude a high-performance logical framework: how to specify, program and verify systems in rewriting logic, Springer-Verlag, 2007.

[29] A. Rodríguez, F. Durán, A. Rutle, L. M. Kristensen, Executing Multilevel Domain-Specific Models in Maude, Journal of Object Technology 18 (2) (2019) 4:1–21. doi:10.5381/jot.2019.18.2.a4.

[30] U. Wolter, F. Macías, A. Rutle, The Category of Typing Chains as a Foundation of Multilevel Typed Model Transformations, Tech. Rep. 2019-417, University of Bergen, Department of Informatics (November 2019).

[31] T. Kühne, A story of levels, in: Proceedings of MULTI @ MODELS, 2018, pp. 673–682.

[32] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2006. doi:10.1007/3-540-31188-2.

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

[34] A. Rodríguez, A. Rutle, L. M. Kristensen, F. Durán, A Foundation for the Composition of Multilevel Domain-Specific Languages, in: MULTI@ MoDELS, 2019, pp. 88–97. doi:10.1109/MODELS-C.2019.00018.

[35] J. de Lara, E. Guerra, Domain-Specific Textual Meta-Modelling Languages for Model Driven Engineering, in: Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings, 2012, pp. 259–274. doi:10.1007/978-3-642-31491-9_20.

[36] A. Wortmann, O. Barais, B. Combemale, M. Wimmer, Modeling languages in Industry 4.0: an extended systematic mapping study, Software and Systems Modeling 19 (1) (2020) 67–94. doi:10.1007/s10270-019-00757-6.

[37] R. M. Burstall, J. A. Goguen, Putting theories together to make specifications, in: Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, USA, August 22-25, 1977, 1977, pp. 1045–1058.

[38] P. Stünkel, H. König, Y. Lamo, A. Rutle, Multimodel correspondence through inter-model constraints, in: S. Marr, J. B. Sartor (Eds.), Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018, ACM, 2018, pp. 9–17. doi:10.1145/3191697.3191715.

[39] P. Stünkel, H. König, Y. Lamo, A. Rutle, Towards multiple model synchronization with comprehensive systems, in: Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Proceedings, Vol. Accepted for publication of Lecture Notes in Computer Science, Springer, 2020.

[40] P. Boehm, H. Fonio, A. Habel, Amalgamation of Graph Transformations: A Synchronization Mechanism, J. Comput. Syst. Sci. 34 (2/3) (1987) 377–408. doi:10.1016/0022-0000(87)90030-4.

[41] G. Taentzer, Parallel and distributed graph transformation - formal description and application to communication-based systems, Berichte aus der Informatik, Shaker, 1996.

[42] E. Biermann, H. Ehrig, C. Ermel, U. Golas, G. Taentzer, Parallel Independence of Amalgamated Graph Transformations Applied to Model Transformation, in: Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday, 2010, pp. 121–140. doi:10.1007/978-3-642-17322-6_7.

[43] J. de Lara Jaramillo, C. Ermel, G. Taentzer, K. Ehrig, Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets, Electron. Notes Theor. Comput. Sci. 109 (2004) 17–29. doi:10.1016/j.entcs.2004.02.053.

[44] Y. Lamo, F. Mantz, A. Rutle, J. de Lara, A declarative and bidirectional model transformation approach based on graph co-spans, in: 15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013, 2013, pp. 1–12. doi:10.1145/2505879.2505900.

[45] A. Rossini, A. Rutle, Y. Lamo, U. Wolter, A formalisation of the copy-modify-merge approach to version control in MDE, J. Log. Algebr. Program. 79 (7) (2010) 636–658. doi:10.1016/j.jlap.2009.10.003.

[46] J. Almeida, A. Rutle, M. Wimmer, Preface to the 6th international workshop on multi-level modelling (MULTI 2019), in: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019, IEEE, 2019, pp. 64–65. doi:10.1109/MODELS-C.2019.00015.

[47] J. P. A. Almeida, A. Rutle, M. Wimmer, T. Kühne, The MULTI Process Challenge, MULTI @MODELS Available at https://bit.ly/2JeDEYi.

[48] E. Gamma, Design patterns: elements of reusable object-oriented software, Pearson Education India, 1995.

[49] J. de Lara Jaramillo, C. Ermel, G. Taentzer, K. Ehrig, Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets, Electron. Notes Theor. Comput. Sci. 109 (2004) 17–29. doi:10.1016/j.entcs.2004.02.053.

[50] L. Bettini, Implementing domain-specific languages with Xtext and Xtend, Packt Publishing Ltd, 2016.

[51] T. Degueule, B. Combemale, A. Blouin, O. Barais, J.-M. Jézéquel, Melange: A meta-language for modular and reusable development of dsls, in: Proceedings of the 2015 SLE Conference, ACM, 2015, pp. 25–36.

[52] J. Jézéquel, B. Combemale, O. Barais, M. Monperrus, F. Fouquet, Mashup of metalanguages and its implementation in the Kermeta language workbench, Software and Systems Modeling 14 (2) (2015) 905–920. doi:10.1007/s10270-013-0354-4.

[53] J.-M. Bruel, B. Combemale, E. Guerra, J.-M. Jézéquel, J. Kienzle, J. de Lara, G. Mussbacher, E. Syriani, H. Vangheluwe, Comparing and classifying model transformation reuse approaches across metamodels, Software and Systems ModelingDoi: 10.1007/s10270-019-00762-9.

[54] G. Perrouin, M. Amrani, M. Acher, B. Combemale, A. Legay, P. Schobbens, Featured model types: towards systematic reuse in modelling language engineering, in: Proceedings of the 8th International Workshop on Modeling in Software Engineering, MiSE@ICSE 2016, Austin, Texas, USA, May 16-17, 2016, 2016, pp. 1–7. doi:10.1145/2896982.2896987.

[55] M. E. Kramer, J. Klein, J. R. H. Steel, B. Morin, J. Kienzle, O. Barais, J. Jézéquel, Achieving Practical Genericity in Model Weaving through Extensibility, in: Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings, 2013, pp. 108–124. doi:10.1007/978-3-642-38883-5_12.

[56] J. Whittle, P. K. Jayaraman, A. M. Elkhodary, A. Moreira, J. Araújo, MATA: A unified approach for composing UML aspect models based on graph transformation, LNCS Trans. Aspect Oriented Softw. Dev. 6 (2009) 191–237. doi:10.1007/978-3-642-03764-1_6.

[57] F. Durán, A. Moreno-Delgado, F. Orejas, S. Zschaler, Amalgamation of domain specific languages with behaviour, Journal of Logical and Algebraic Methods in Programming 86 (2017) 208–235. doi:https://doi.org/10.1016/j.jlamp.2015.09.005.

[58] A. Rensink, J. Kuperus, Repotting the Geraniums: On Nested Graph Transformation Rules, ECEASST 18. doi:10.14279/tuj.eceasst.18.260.

[59] D. Balasubramanian, A. Narayanan, S. Neema, F. Shi, R. Thibodeaux, G. Karsai, A Subgraph Operator for Graph Transformation Languages, ECEASST 6. doi:10.14279/tuj.eceasst.6.72.

Appendix A. Complete process management multilevel hierarchy

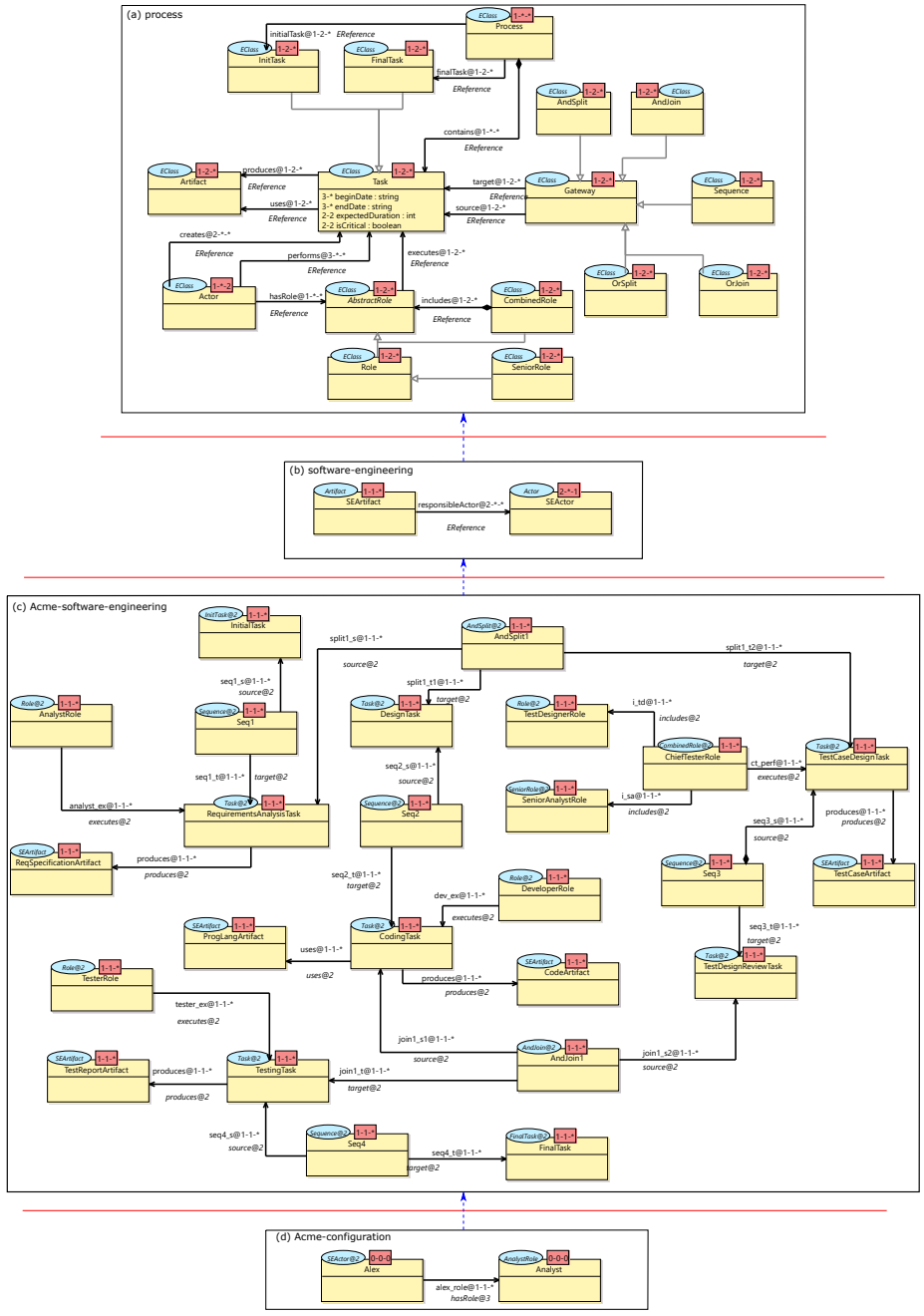


Figure A.41: Process management multilevel hierarchy

Appendix B. Amalgamated MCMT rules computed in MultEcore

```
rule CreateTaskUndertakeActivity{
        meta{
                //Nodes level 1 - Process
                Role: $process[1]!Role
                Actor: $process[1]!Actor
                Task: $process[1]!Task
                //Nodes level 1 - Human
                Human: $human[1]!Human
                        Human.stamina : Integer
                Activity: $human[1]!Activity
                        Activity.impact : Integer
                //Edges level 1 - Process
                hasRole: $process[1]!Actor.hasRole
                performs: $process[1]!Actor.performs
                executes: $process[1]!Role.executes
                //Edges level 1 - Human
                does: $human[1]!Human.does
                //Nodes level 2 - Process
                R1: process[2]!Role
                T1: process[2]!Task
                //Nodes level 2 - Human
                Worker: human[2]!Human
                        Worker.profit : Integer
                Assignment: human[2]!Activity
                        Assignment.value : Integer
                //Edges level 2 - Process
                e: process[2]!Role.executes
                //Edges level 2 - Human
                undertakes: human[2]!Human.does
                //Source.edge = Target

                [Actor.hasRole = Role]
                [Actor.performs = Task]
                [Role.executes = Task]
                [Human.does = Activity]
                [R1.e = T1]
                [Worker.undertakes = Assignment]
        }
        from {
                act1work1: Actor, Worker
                        act1work1.stamina = #s#
                        act1work1.profit = #p#
                r1: R1
                task1as1: EClass, Assignment
                        task1as1.impact = #i#
                        task1as1.value = #v#
                a1role: hasRole

                [act1work1.a1role = r1]
        }
        to {
                act1work1: Actor, Worker
                        act1work1.stamina = #s - i#
                        act1work1.profit = #p + v#
                r1: R1
                task1as1: T1, Assignment
                        task1as1.impact = #i#
                        task1as1.value = #v#
                a1role: hasRole
                a1pu: performs, undertakes
                r1e: e

                [act1work1.a1role = r1]
                [act1work1.a1pu = task1as1]
                [r1.r1e = task1as1]
        }
}
```

Figure B.42: Full CreateTaskUndertakeActivity MCMT rule computed in MultEcore. It corresponds to MCMT rule depicted in Figure 33

```
rule ProduceArtefactkUndertakeActivity{
        meta{
                //Nodes level 1 - Process
                Actor: $process[1]!Actor
                Task: $process[1]!Task
                Artifact: $process[1]!Artifact
                //Nodes level 1 - Human
                Human: $human[1]!Human
                         Human.stamina : Integer
                Activity: $human[1]!Activity
                         Activity.impact : Integer
                //Edges level 1 - Process
                performs: $process[1]!Actor.performs
                produces: $process[1]!Task.produces
                //Edges level 1 - Human
                does: $human[1]!Human.does
                //Nodes level 2 - Process
                SEActor: process[2]!Actor
                SEArtifact: process[2]!Artifact
                //Nodes level 2 - Human
                Worker: human[2]!Human
                         Worker.profit : Integer
                Assignment: human[2]!Activity
                         Assignment.value : Integer
                //Edges level 2 - Process
                responsibleActor: process[2]!EReference
                //Edges level 2 - Human
                undertakes: human[2]!Human.does
                //Nodes level 3 - Process
                T1 : process[3]!Task
                A1 : process[3]! SEArtifact
                                //Edges level 3 - Process
                p1 : process[3]!Task.produces

                //Source.edge = Target
                [Actor.performs = Task]
                [Task.produces = Artifact]
                [Human.does = Activity]
                [SEArtifact.responsibleActor = SEActor]
                [Worker.undertakes = Assignment]

                [T1.p1 = A1]
        }
        from {
                act1work1: SEActor, Worker
                         act1work1.stamina = #s#
                         act1work1.profit = #p#
                task1as1: T1, Assignment
                         task1as1.impact = #i#
                         task1as1.value = #v#
                a1pu: performs, EReference

                [act1work1.a1pu = task1as1]
        }
        to {
                act1work1: SEActor, Worker
                         act1work1.stamina = #s - i#
                         act1work1.profit = #p + v#
                task1as1: T1, Assignment
                         task1as1.impact = #i#
                         task1as1.value = #v#
                ar1 : A1
                a1pu: performs, undertakes
                t1pr : p1
                r : responsibleActor

                [act1work1.a1pu = task1as1]
                [task1as1.t1pr = ar1]
                [ar1.r = act1work1]
        }
}
```

Figure B.43: Full ProduceArtefactUndertakeActivity MCMT rule computed in MultEcore. It corresponds to MCMT rule depicted in Figure 34