Language family engineering with product lines of multi-level models

Juan de Lara[1] and Esther Guerra[1]

[1]Modelling and Software Engineering Research Group, Computer Science Department, Universidad Autónoma de Madrid, Madrid, Spain

Abstract. Modelling is an essential activity in software engineering. It typically involves two meta-levels: one includes meta-models that describe modelling languages, and the other contains models built by instantiating those meta-models. *Multi-level modelling* generalizes this approach by allowing models to span an arbitrary number of meta-levels. A scenario that profits from multi-level modelling is the definition of language families that can be specialized (e.g., for different domains) by successive refinements at subsequent meta-levels, hence promoting language reuse. This enables an *open* set of variability options given by all possible specializations of the language family. However, multi-level modelling lacks the ability to express *closed* variability regarding the availability of language primitives or the possibility to opt between alternative primitive realizations. This limits the reuse opportunities of a language family. To improve this situation, we propose a novel combination of product lines with multi-level modelling to cover both open and closed variability. Our proposal is backed by a formal theory that guarantees correctness, enables top-down and bottom-up language variability design, and is implemented atop the METADEPTH multi-level modelling tool.

Keywords: Meta-modelling, Multi-level modelling, Product lines, Domain-specific languages, Software language engineering, METADEPTH.

1. Introduction

Modelling is intrinsic to most engineering disciplines. Within software engineering, it plays a pivotal role in model-driven engineering (MDE) [Sch06]. This is a software construction paradigm where models are actively used to describe, analyse, validate, verify, synthesize and maintain the application to be built, among other activities [BCW17].

Models are built using modelling languages, which can be either general-purpose, like the UML [UML17], or domain-specific languages (DSLs) tailored to a specific concern [KT08, VBD+13]. In MDE, the abstract syntax of modelling languages is defined through a meta-model that describes the primitives that can be used in models one meta-level below. This modelling approach, which is the standard nowadays, constrains engineers to confine their models within one meta-level (the "model" level).
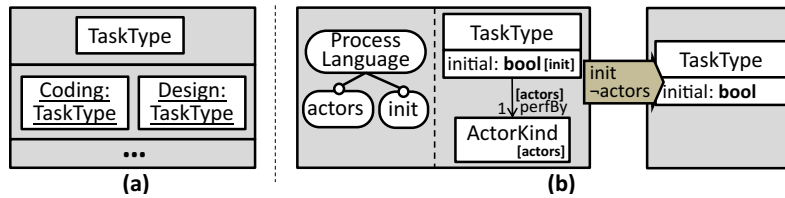

Fig. 1. a Open language variability through instantiation. **b** Closed language variability through product lines.

Several researchers have observed that domain modelling can benefit from the use of more than one meta-level [AK08, dLGS14, FAGdC18, Fra14, IGSS18, GPHS06, MRS⁺18]. This way of modelling—called multi-level modelling [AK01] or deep meta-modelling [dLG10]—yields simpler models in scenarios that involve the type-object pattern [AK08, dLGS14, MRB97]. Moreover, it permits defining language families (e.g., for process modelling) that can be specialized to specific domains (e.g., software process modelling, industrial process modelling) via instantiation at lower meta-levels [dLGC15]. Instantiation is an *open* variability mechanism that permits the customization of a language by specializing its primitives for a domain, or adding new ones via so-called linguistic extensions [dLG10]. As an illustration, Fig. 1a shows a tiny process modelling language that defines the primitive TaskType, which is customized by instantiation in the lower meta-level for the software process modelling domain (Coding and Design). In turn, these two primitives could be instantiated in the meta-level below. However, multi-level modelling lacks support for expressing optionality of language primitives or alternative primitive realizations. This prevents wider language reuse and customization possibilities.

Software product lines (SPLs) encompass methods, tools and techniques to engineer collections of similar software systems using a common means of production [NC02, PBL05]. SPLs support *closed* variability, where a concrete software product is obtained by selecting among a finite set of available features (i.e., by setting a *configuration*). SPL techniques have been applied to language engineering to define product lines of languages representing a closed set of predefined language variants [GdLCS20, PAA⁺16, WHG⁺09]. As an example, Fig. 1b shows a process modelling language product line with two configurable features: actors and initial tasks. Selecting a configuration of features (in the figure, initial tasks but no actors) yields a language variant. Languages defined via a product line permit configuring the language primitives and their realization, but cannot be specialized for specific domains, because this requires from open variability mechanisms.

To improve current language reuse techniques, we propose combining multi-level modelling and product lines. This allows the definition of highly configurable language families that profit from both open variability (as given by instantiation) and closed variability (as given by configuration). This way, this paper makes the following contributions: (i) a novel notion of multi-level model product line; (ii) a theory that enables deferring variability resolution to lower meta-levels in a flexible way, guaranteeing the correctness of interleavings of instantiation and configuration steps; (iii) techniques supporting both top-down and bottom-up variability design, based on the possibility of advancing variability extension to both instantiation and configuration; and (iv) an implementation of these ideas on top of the METADEPTH tool [dLG10].

This work builds on our FASE'20 article [dLG20], expanding it in three main ways, covering both *usage* and *design* of language product lines. First, we complete the presented theory with required definitions and lemmas for composition of specializations steps (Definition 3 and Lemma 1; Definition 9 and Lemma 2; Definition 12 and Lemma 3). Then, we expand the theory to calculate the fully configured language (i.e., a language definition with no variability) that is equivalent to the language resulting from an arbitrary chain of language instantiations and partial configurations, as shown by Theorem 5.3. This is an important result, which shows that, in order to use a language family, users do not need to provide a full configuration before using the family. Instead, they can directly use it by instantiation, while variability can be resolved at later steps providing partial configurations as needed. The second main extension of this paper facilitates designing language product lines. In [dLG20], we assumed that a language family needed to be constructed in a top-down way, where all variability is designed up-front. This was due to the fact that the theory did not have a way to characterize *extensions* of the feature models and the associated model product lines (instead, the theory only covered specializations). However, in practice, product lines can be constructed both top-down and bottom-up [KC16]. Supporting these two options in our setting requires enabling exploratory modelling, where the language is instantiated, possibly partially specialized, and then new variants can be added, which the designer might like to include in the original language definition. For this purpose, we introduce new notions of extension morphisms (Definitions 14 and 15), along with compatibility conditions ensuring that variability extensions can be advanced to both instantiation and specialization (Lemma 4, Theorem 5.4 and Corollary 1). Accordingly, we use the new concepts and the theory

to describe flexible processes for using language families (Sect. 5.1), and for their top-down and bottom-up construction (Sect. 5.2). As a third main extension, we have expanded the tool with new functionality to support extension, and provide a walkthrough of its use for three activities: top-down creation of language families (Sect. 6.1), use of language families (Sect. 6.2) and bottom-up extension of language families (Sect. 6.3). Finally, we provide additional examples and explanations, as well as a more thorough comparison with related work.

The rest of this paper is organized as follows. Section 2 introduces multi-level modelling and identifies the challenges tackled in this paper. Section 3 provides a light formalization of multi-level modelling, which is extended with product line techniques in Sect. 4. Section 5 exploits the introduced concepts for: (a) the flexible use of language families, by proving that variability configuration can be deferred to model instantiation; and (b) the bottom-up and top-down construction of language families, by showing that variability extension can be advanced to both configuration and instantiation. Section 6 describes tool support. Section 7 discusses related research, and Sect. 8 ends with the conclusions and future work. An appendix includes the proofs of the theorems and lemmas in the paper.

2. Multi-level modelling: intuition and challenges

In this section, we first introduce the main concepts of multi-level modelling guided by an example (Sect. 2.1), and then we discuss some challenges when applying multi-level modelling to language engineering (Sect. 2.2).

2.1. Multi-level modelling by example

Multi-level modelling supports the definition of models using multiple meta-levels [AK08, dLGS14]. To understand its rationale, assume we would like to create a language to define commerce information systems (a standard example often used in the multi-level modelling literature [AK08, dLGS14]). The language should allow defining *product types* (like *books* or *food*) which have a *tax*, as well as *products* of the defined types (like *Othello* or *banana*) which have a *price*. Moreover, some product types may need to define specific properties, like the number of *pages* in books.

Figure 2a shows a solution for this scenario using two meta-levels. In this solution, the meta-model uses the *type-object* pattern [MRB97] to emulate the typing relationship between Product and ProductType. In addition, classes Attribute and Slot permit defining properties in ProductTypes and assigning them a value in Products (called *dynamic features* pattern in [dLGS14]). The model in the bottom meta-level represents an information system for Kiosks. It defines the product types Book and Food, as well as the products sold by a particular kiosk: the Othello book and Bananas.

On reflection, one can realize that this solution emulates two meta-levels within one, as we convey with the dashed line in Fig. 2a. Therefore, we show an alternative multi-level solution using three meta-levels in Fig. 2b. The top level defines just ProductType, which is instantiated at the next level to create Book and Food product types, which in turn are instantiated at the bottom level to create specific products. Hence, elements in this approach are uniformly called *clabjects* [Atk97] across meta-levels (from the contraction of the words *cla*ss and o*bject*), since they are types for the elements in the level below, and instances of the elements in the level above. For example, clabject Book is a type for Othello and an instance of ProductType. This multi-level solution leads to a simpler model (with fewer elements) because a clabject suffices to represent both ProductType and Product.

Clabjects may need to control the properties of their instances beyond the next meta-level. For example, the direct instances of ProductType need to have a tax, and the instances of its instances (which we call *indirect* instances) have a price. This is possible by the use of a *deep characterization* mechanism called *potency* [Atk97, AK01]. The potency is a natural number, or zero, which governs the instantiation depth of models, clabjects and features. Fig. 2b depicts the potency after the "@" symbol, and the elements that do not declare potency take it from their container. As an example, attribute ProductType.price takes its potency from ProductType, and this from the Commerce model, which declares potency 2. When an element is instantiated, the instance receives the potency of the element minus 1. Elements with potency 0 are pure instances and cannot be instantiated. For example, attribute ProductType.tax with potency 1 is instantiated into Book.tax and Food.tax, which therefore have potency 0 and can receive values. As model Commerce has potency 2, it can be instantiated at the two subsequent meta-levels. The potency of a model is often called its *level* [AK08].

Sometimes, it is not possible to foresee every possible property required by clabject instances several meta-levels below—like the number of pages in books—or we may need to introduce new primitives at lower levels—like a new clabject to model the authors of books.
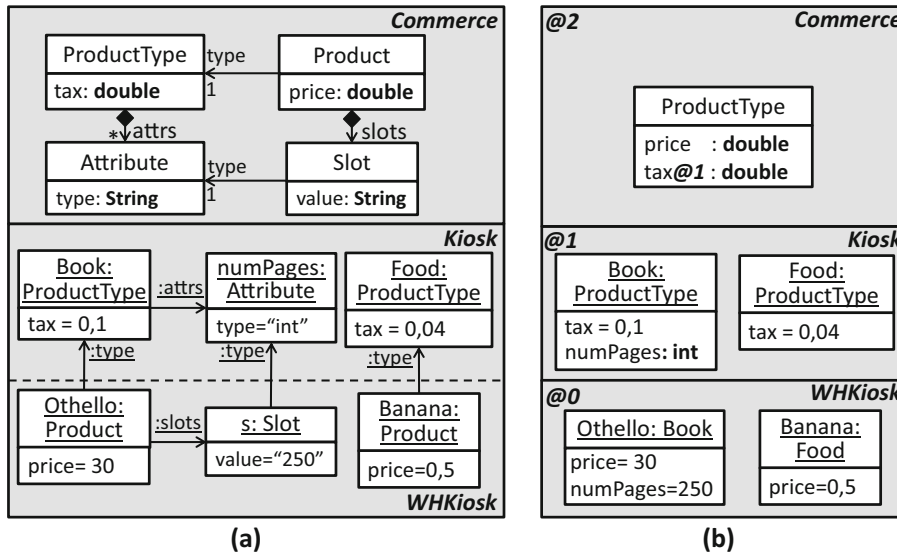
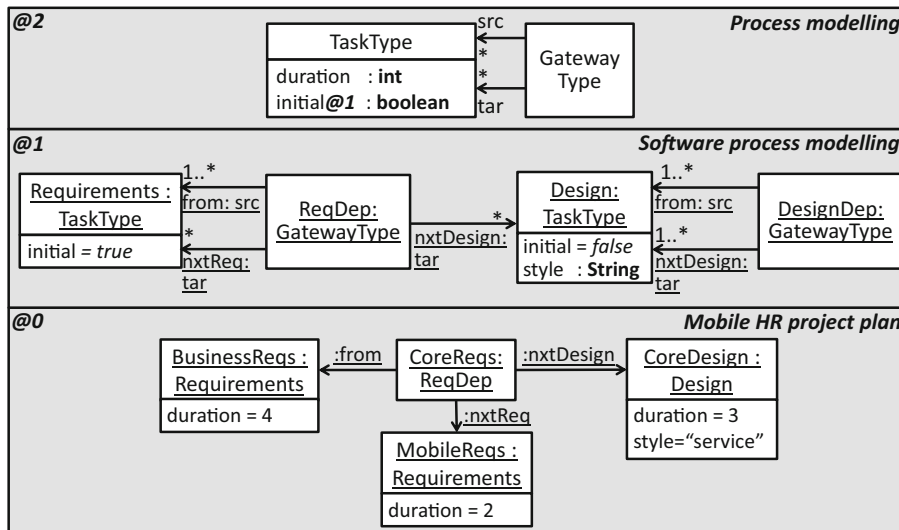**Fig. 2.** Commerce example using **a** standard modelling and **b** multi-level modelling.

Fig. 3. Multi-level model for process modelling, and application to software process modelling.

To handle those cases, multi-level modelling supports *linguistic extensions* [dLGC15]. These are elements (clabjects or features) with no ontological type, but with a linguistic type that corresponds to the meta-modelling primitive used to create it (see Orthogonal Classification Architecture in [AK02] for more details). As an example, Book.numPages is a linguistic extension modelling a property specific to Book but not to other product types. Instead, in the two-level solution in Fig. 2a, the properties of specific ProductTypes need to be explicitly modelled by classes Attribute and Slot, leading to more complexity.

2.2. Improving reuse in multi-level modelling: some challenges

Multi-level modelling enables language reuse by supporting the definition of language families. For example, Fig. 3 shows at the top a generic process modelling language that can be used to define process modelling languages for different domains, like education, software engineering, or production engineering. The language is designed to consider three levels. Level 2 at the top contains the language definition, consisting of primitives (i.e., clabjects) to define task and gateway types. Level 1 contains language specializations for specific domains.
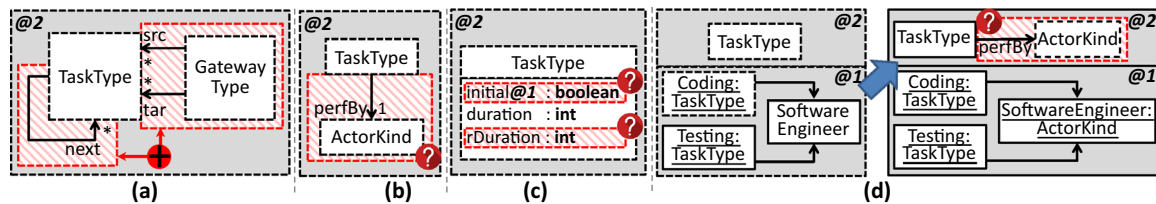
Fig. 4. Examples of variability needs: **a** alternative primitive realizations, **b** optional primitives, **c** optional attributes, **d** bottom-up variability design.

The figure shows the case for the software engineering domain, which defines the task types Requirements and Design, and two gateway types: ReqDep to transition from requirement tasks to either design or requirement tasks, and DesignDep to declare dependencies between design tasks. Finally, level 0 contains domain-specific processes. The one in the figure declares two requirements tasks, one design task and one gateway.

This example shows how instantiation permits customizing the language primitives offered at the top level for particular domains, and how linguistic extensions (e.g., attribute Design.style at level 1 in Fig. 3) allow adding domain-specific primitives and properties to language specializations. However, the following scenarios require further facilities that enable a better fit for particular domains, increase language reuse and facilitate language family engineering.

- **Alternative realizations** A language primitive may be realised in different ways, each more adequate than the others depending on the domain. For example, in Fig. 3, dependencies between task types are modelled by Gateway-Type. However, in domains that do not require distinguishing types of gateways or n-ary dependencies, a simpler representation of dependencies as a binary reference between TaskTypes is enough (see Fig. 4a). Unfortunately, multi-level modelling does not support this kind of variability, which enables alternative realizations of available language primitives.

- **Primitive excess** Some offered language primitives may be unnecessary in simple domains. This can be controlled by not instantiating the primitive. However, withdrawing the needless primitives may be a better option because it simplifies the language usage and avoids some problematic situations. First, if the needless primitive is an attribute (like initial in Fig. 4), then it becomes instantiated by force, polluting the model with unnecessary information. Second, some mandatory primitives may not be needed in certain domains. For example, in Fig. 4b, the language designer assumes that any TaskType (e.g., Requirements) will be performed by one ActorKind (e.g., Analyst or DomainExpert). However, there may be domains that do not involve actors (e.g., if tasks are automated), but the mandatory relation perfBy still forces having instances of ActorKind associated to instances of TaskType.

- **Deferred variability resolution and exploratory modelling** The decision about the inclusion or not of a primitive may not be clear when the language is instantiated for a domain, but this can be determined later at lower meta-levels. For example, in Fig. 4c, an engineer might hesitate whether, in addition to the expected task duration (attribute duration), tasks should store their real duration as well (attribute rDuration with potency 2); in such a case, the engineer may prefer deferring the decision to level 1 or 0, where the need becomes evident. In general, resolving all variability in a language family at the top level may be hasty in some cases, since the suitability of a primitive may be apparent only when a language has reached certain specificity (i.e., at lower meta-levels). Moreover, enabling modelling before resolving every possible language variability option may be good for exploratory purposes.

- **Top-down and bottom-up variability design** Language variability can be designed up-front, following a *top-down* process. However, some language variants may emerge when working in a specific domain, making it desirable to lift the discovered variant *bottom-up* [CdLG12] from a lower meta-level to the top one. Fig. 4d shows an example. The left side shows a process modelling language that does not support actors (level 2), but its refinement for software engineering requires software engineers—a kind of actor—so it defines the linguistic extension SoftwareEngineer (level 1). Since other domains may need to support actors, SoftwareEngineer could be lifted to the top level renamed more generically to ActorKind, as the right side of Fig. 4d shows. ActorKind would be optional as not every domain needs actors. This kind of bottom-up refactorings [dLG18] would facilitate extending the definition and variability of language families upon emerging variants. Supporting both top-down and bottom-up variability design would provide flexibility to language family creation and evolution.
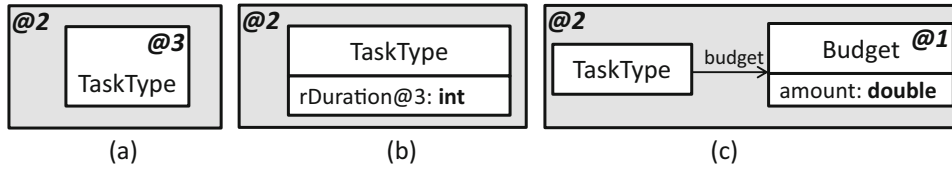
Fig. 5. Illustrating the violation of the well-formedness rules in Definition 1 for potency. **a** A clabject with higher potency than the model level. **b** A slot with higher potency than the one of its container clabject. **c** A reference with higher potency than its target clabject.

To tackle these challenges, we incorporate variability into multi-level models taking ideas from SPLs. As a first step, in the next section we formalize multi-level models.

3. A formal foundation for multi-level modelling

We start defining the structure of models equipped with deep characterization, which we call *deep* models. We represent models at different meta-levels in a uniform way, in order to cope with an arbitrary number of meta-levels. For simplicity of presentation, and because they are not essential to demonstrate our ideas, we omit inheritance, cardinalities and integrity constraints in our formalization.

Def. 1 (Deep model). A deep model is a tuple $M = \langle p, C, S, R, src, tar, pot \rangle$, where:

- $p \in \mathbb{N}_0$ is called the model potency, or *level*.
- C, S and R are disjoint sets of clabjects, slots and references, respectively.
- $src \colon S \cup R \to C$ is a function assigning the owner clabject to slots and references.
- $tar \colon R \to C$ is a function assigning the target clabject to each reference.
- $pot \colon C \cup S \cup R \to \mathbb{N}_0$ is a function assigning a potency to clabjects, slots and references s.t.:

  1. $\forall\, e \in C \cup S \cup R \bullet pot(e) \leq p$
  2. $\forall\, s \in S \cup R \bullet pot(s) \leq pot(src(s))$
  3. $\forall\, r \in R \bullet pot(r) \leq pot(tar(r))$

In the previous definition, we assign a level p to deep models. Elements in a deep model have a potency via function pot, which must satisfy three conditions: (1) the potency of an element should not be larger than the model level, (2) the potency of slots and references should not be larger than the one of their container clabject, and (3) the potency of references should not be larger than the one of the clabjects they point to. Please note that we use the term *slot* to refer uniformly to attributes (or their instances) at any meta-level.

Example Figure 5 illustrates the rationale of the three well-formedness rules in Definition 1 concerning potency. Specifically, each deep model violates one of the rules. Figure 5a depicts a deep model where the level (2) is lower than the potency of its contained clabject (3). This would be problematic two levels below, where the indirect instances of TaskType have potency 1 but cannot be instantiated because their container model has potency 0 and hence is non-instantiable. Similarly, Fig. 5b shows a slot with higher potency (3) than its container clabject (2). In this case, the indirect instances of the clabject with potency 0 will contain a slot rDuration with potency 1. The slot would receive a value one level below, when it reaches potency 0, but this is not possible because its container clabject cannot be instantiated further. Finally, in Fig. 5c, reference budget with potency 2 points to clabject Budget with potency 1. As a consequence, the reference can only be instantiated at the next level regardless its potency 2, due to the lower potency of Budget.

Next, we define a general notion of mapping (a *morphism*) between deep models as a tuple of three (total) functions between the sets of clabjects, slots and references. Each morphism has a *depth* (a natural number or 0) controlling the distance between the levels of the involved models. We use two particular types of mappings to represent the *type* relation between deep models at adjacent meta-levels (when the morphism depth is 1), and *extensions* of a deep model to add linguistic extensions (when the depth is 0).

Def. 2 (D-morphism, type and extension). Given two deep models $M_i = \langle p_i, C_i, S_i, R_i, src_i, tar_i, pot_i \rangle$ for $i = \{0, 1\}$, a deep model morphism (D-morphism in short) $m = \langle d, m_C, m_S, m_R \rangle \colon M_0 \to M_1$ is a tuple made of a number $d \in \mathbb{N}_0$ called *depth*, and three functions $m_C \colon C_0 \to C_1$, $m_S \colon S_0 \to S_1$ and $m_R \colon R_0 \to R_1$ s.t.:

$$S_1 \xrightarrow{src_1} C_1 \qquad\qquad C_1 \xleftarrow{src_1} R_1 \xrightarrow{tar_1} C_1$$

$$m_S \uparrow \quad = \quad \uparrow m_C \qquad\qquad m_C \uparrow \quad = \quad \uparrow m_R \quad = \quad \uparrow m_C$$

$$S_0 \xrightarrow{src_0} C_0 \qquad\qquad C_0 \xleftarrow{src_0} R_0 \xrightarrow{tar_0} C_0$$

Fig. 6. Commutativity conditions for D-morphisms in Definition 2.

1. $p_0 + d = p_1$
2. $\forall\, e \in X_0 \bullet pot_0(e) + d = pot_1(m_X(e))$ (for $X \in \{C, S, R\}$)
3. Each function m_C, m_S, m_R commutes with functions src_i and tar_i (see Fig. 6)

D-morphism $tp = \langle d, tp_C, tp_S, tp_R \rangle \colon M_0 \to M_1$ is called *type* if $d = 1$, and is called *indirect type* if $d > 1$. M_1 is called the (indirect) model type of M_0.

D-morphism $ex = \langle d, ex_C, ex_S, ex_R \rangle \colon M_0 \to M_1$ is called *level-preserving* if $d = 0$. A level-preserving D-morphism ex is called *extension* if each ex_X (for $X = \{C, S, R\}$) is an inclusion. An extension is called *identity* if each ex_X is surjective.

In Definition 2, condition 1 ensures that the D-morphism connects models of suitable levels (at a distance of d levels), condition 2 checks that the potency of elements in M_0 decreases according to the depth of the D-morphism, and condition 3 ensures that the D-morphism is coherent with the source and target of slots and references (just like in standard graph morphisms [EEPT06]). We use total functions to represent the type, which ensures that each element in a deep model has a type. Linguistic extensions are not typed, but they are modelled as an extension D-morphism of a (typed) deep model into a larger model. This avoids resorting to partial functions to represent the type, which would complicate the formalization [RdLG+14, WMR20]. Identity extensions map isomorphic deep models.

D-morphisms can be composed by composing the three mappings and adding their depths, as the next definition and lemma show. Composition of D-morphisms is necessary as a basis for the results of Sect. 5.

Def. 3 (D-morphism composition). Given two D-morphisms $m_1 = \langle d^1, m_C^1, m_S^1, m_R^2 \rangle \colon M_0 \to M_1$ and $m_2 = \langle d^2, m_C^2, m_S^2, m_R^2 \rangle \colon M_1 \to M_2$, the composed morphism $m_2 \circ m_1 \colon M_0 \to M_2$ is defined as $\langle d^1 + d^2, m_C^2 \circ m_C^1, m_S^2 \circ m_S^1, m_R^2 \circ m_R^1 \rangle$.

Lemma 1 (D-morphism composition yields a D-morphism). Given two D-morphisms $m_1 = \langle d^1, m_C^1, m_S^1, m_R^2 \rangle \colon M_0 \to M_1$ and $m_2 = \langle d^2, m_C^2, m_S^2, m_R^2 \rangle \colon M_1 \to M_2$, their composition $m_2 \circ m_1 \colon M_0 \to M_2$ is a valid D-morphism.

Proof. By compositionality of functions over sets. See proof details in "Appendix". □

Remark The composition of two (indirect) type D-morphisms is an indirect type D-morphism. The composition of two level-preserving D-morphisms is level preserving, and it is an extension (resp. identity) if both D-morphisms are extensions (resp. identities).

A multi-level model is made of a root deep model, and a sequence of pairs of instantiations and extensions. The length of this sequence is equal to the root model level. The extensions are allowed to be identity extensions.

Def. 4 (Multi-level model). A multi-level model $MLM = \langle M'_0, ML = \langle (M'_i \xleftarrow{tp_{i+1}} M_{i+1} \xrightarrow{ex_{i+1}} M'_{i+1}) \rangle_{i=0..p'_0-1} \rangle$ is made of a deep model M'_0 called the *root*, and a sequence ML of length p'_0 (the level of M'_0) of spans of D-morphisms, where the left D-morphism is a *type* and the right D-morphism a (possibly identity) *extension*.

Example Figure 7 shows a multi-level model (a small excerpt of the one in Fig. 3) according to Definition 4. Slots are represented as rounded nodes, instead of inside the owner clabject box. Figure 3 hides the slots with potency bigger than 0 that are typed, like Design.duration at level 1, but such instances do exist and are explicitly shown in Figure 7 (see slot duration' in models M₁ and M'₁). The figure shows a clabject TaskType in the root model M'₀, its instance called Design in model M₁, a subsequent extension that adds a style slot to Design (model M'₁), an instantiation of it (model M₂), and an identity extension (model M'₂). Whenever a model does not include linguistic extensions, like M₂, we use the identity extension D-morphism. Since slot initial' in model M'₁ has potency 0, it is not instantiated in the model with level 0 (M₂). Finally, it would be possible to derive the (indirect) type of M₂ with respect to M'₀ by defining a construction akin to a pullback (in a category made of multi-level models and D-morphisms) that yields the part of M₂ typed by M₁ [Lan71].
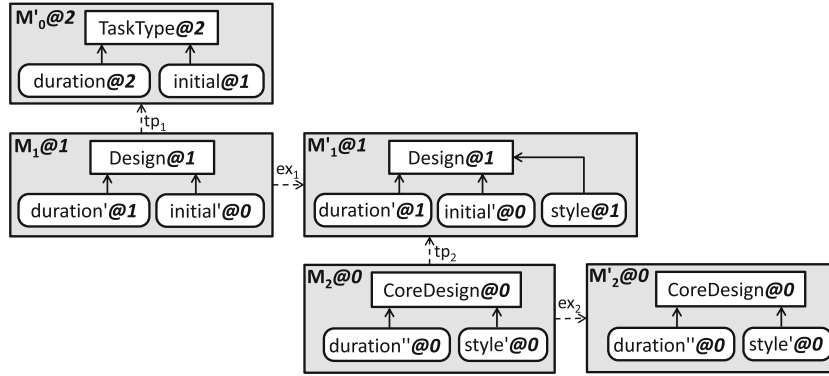
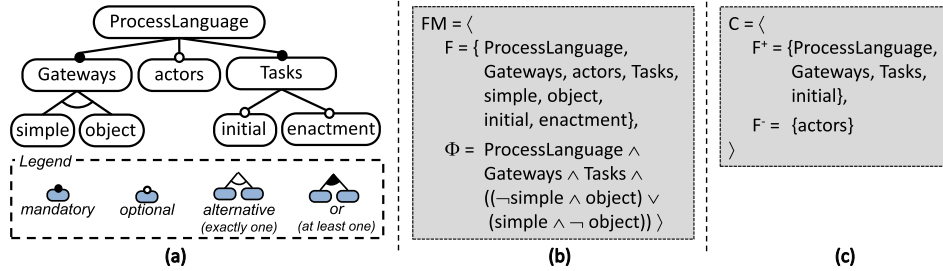**Fig. 7.** Multi-level model example, according to Definition 4.

Fig. 8. a Feature model for the running example using the feature diagram notation. **b** Feature model for the running example using Definition 5. **c** A partial configuration.

4. Multi-level model product lines

In order to solve the challenges identified in Sect. 2.2, we extend deep models with closed variability options by borrowing concepts from product lines. We use feature models [KCH+90] to represent the allowed variability.

Def. 5 (Feature model). A feature model $FM = \langle F, \Phi \rangle$ consists of a set F of propositional variables called features, and a satisfiable propositional formula Φ over the variables in F, specifying the valid feature configurations.

Example Figure 8 shows the feature model for the running example using (a) the standard feature diagram notation [KCH+90], and (b) Definition 5. The feature model permits choosing if the process modelling language will have primitives to define actors (feature actors, cf. Fig. 4b), initial tasks and their enactment at level 0 (features initial and enactment, cf. Fig. 4c), as well as selecting whether gateways are to be represented either as references or objects (features simple and object, cf. Fig. 4a). The feature model includes the mandatory features ProcessLanguage, Gateways and Tasks as syntactic sugar to obtain a tree representation, but they are not needed in our formalization.

The selection of one option within the variability space offered by a feature model is done through a *configuration*. A configuration specifies sets of selected and discarded features, assigning the value *true* to the former and *false* to the latter. To enhance flexibility of use, we also support *partial* configurations where some features are not given any value (i.e., they are neither selected nor discarded). We will use partial configurations to allow deferring the resolution of some variability options to lower meta-levels.

Def. 6 (Configuration). Given a feature model $FM = \langle F, \Phi \rangle$, a configuration of FM is a tuple $C = \langle F^+, F^- \rangle$ made of two disjoint sets $F^+ \subseteq F$ and $F^- \subseteq F$, s.t. $\Phi[F^+/true, F^-/false] \not\cong false$. C is *total* if $F = F^+ \cup F^-$, otherwise it is *partial*. The set of all configurations of FM is denoted by $CFG(FM)$.

Given two configurations $C_i = \langle F_i^+, F_i^- \rangle \in CFG(FM)$ (for $i = \{0, 1\}$), C_0 is smaller than or equal to C_1, written $C_0 \le C_1$, if $F_0^+ \subseteq F_1^+$ and $F_0^- \subseteq F_1^-$. Similarly, $C_0 < C_1$ if $C_0 \le C_1$ and either $F_0^+ \subset F_1^+$ or $F_0^- \subset F_1^-$.

In the previous definition, F^+ contains the selected features (i.e., those given the value *true*), F^- the discarded features (i.e., those given the value *false*), and $F \setminus (F^+ \cup F^-)$ is the set of features whose value has not been set. A configuration must be compatible with the feature model, so Definition 6 demands that the formula Φ of the feature model is not false after substituting the features in F^+ by *true* and the features in F^- by *false*. If the configuration is total, then the condition entails that Φ must evaluate to *true*. The relation $<$ between configurations defines a partial order where total configurations are maximal elements, and the empty configuration (i.e., the configuration that does not select or discard any feature) is the minimal element.

Remark We sometimes use the term *invalid configuration* for a tuple $C = \langle F^+, F^- \rangle$ with $F^+ \subseteq F$ and $F^- \subseteq F$, s.t. $\Phi[F^+/true, F^-/false] \cong false$.

Example Figure 8c shows an example of configuration, which selects features ProcessLanguage, Gateways, Tasks and initial; and discards the feature actors. Since features simple, object and enactment remain undefined, it is a partial configuration. The result from substituting the selected and discarded features by their values in the feature model formula is the following: $\Phi[\{\text{ProcessLanguage, Gateways, Tasks, initial}\}/true, \{\text{actors}\}/false] \cong$ ($\neg$simple$\wedge$object) $\vee$ (simple$\wedge\neg$object)).

Next, we assign a level to feature models, and potencies to features, in order to control the level at which features should be assigned a truth value.

Def. 7 (Deep feature model). A *deep feature model* $DFM = \langle l, FM = \langle F, \Phi \rangle, pot \rangle$ is made of a level $l \in \mathbb{N}_0$, a feature model FM, and a function $pot: F \to \mathbb{N}_0$ assigning a potency to each feature, s.t. $\forall f \in F \bullet pot(f) \leq l$.

Next, we define a mapping between deep feature models, called F-morphism. Similar to D-morphisms (cf. Definition 2), F-morphisms have a depth that can be positive or 0. In addition, they include a configuration, and a mapping for the features excluded from the configuration (i.e., those without a value). This is necessary, since we want to represent specialization relations between the two feature models [TBK09] by means of the (partial) configuration of the morphism. We identify two special kinds of F-morphisms: one representing a type relationship between two feature models (where the morphism depth is 1 and the configuration empty), and the other expressing a specialization relationship between two feature models via a total or partial configuration (where the morphism depth is 0).

Def. 8 (F-morphism, type and specialization). Given two deep feature models $DFM_i = \langle l_i, FM_i, pot_i \rangle$ (for $i = \{0, 1\}$), a deep feature model morphism (F-morphism in short) $m = \langle d, m_F, C \rangle : DFM_0 \to DFM_1$ is made of:

- a depth $d \in \mathbb{N}_0$ s.t. $l_0 + d = l_1$
- an injective set morphism $m_F: F_0 \to F_1$ s.t. $\forall f \in F_0 \bullet pot_0(f) + d = pot_1(m_F(f))$
- a configuration $C = \langle F_1^+, F_1^- \rangle \in CFG(FM_1)$ s.t.:

  1. $m_F(F_0) = F_1 \setminus (F_1^+ \cup F_1^-)$
  2. $\Phi_1[F_1^+/true, F_1^-/false] \cong \Phi_0[F_0/m_F(F_0)]$

F-morphism tp is a *type* morphism if $d = 1$ and $C = \langle \emptyset, \emptyset \rangle$, and it is an *indirect type* morphism if $d > 1$ and $C = \langle \emptyset, \emptyset \rangle$. F-morphism sp is a *specialization* if $d = 0$.

Definition 8 requires that the F-morphism depth fills the gap between the feature model levels, and between the potencies of the mapped features. FM_0 may have fewer features than FM_1, in case the configuration C assigns a value to the missing features with respect to FM_1. In particular, the injectivity condition of m_F and requiring $m_F(F_0) = F_1 \setminus (F_1^+ \cup F_1^-)$ ensures that only the features left undefined by C are mapped from FM_0. Moreover, when the configuration C assigns a value to some feature, the definition requires that the formula Φ_1 after replacing the features in C by their value *true* or *false*, is equivalent to Φ_0 after replacing the features in F_0 by their mapping in F_1. This corresponds to a (partial) evaluation of the formula Φ_1 as a result of a feature model specialization.

Example Figure 9 shows two F-morphisms, with tp a type and sp a specialization. F-morphism $tp: FM_1 \to FM_2$ relates two deep feature models FM_1 and FM_2, where the level and potencies of FM_1 are one less than those in FM_2, and the formulae are the same modulo feature renaming. Specialization $sp: FM_0 \to FM_1$ has depth 0 and partial configuration $C = \langle F^+ = \{\text{object}\}, F^- = \{\text{simple}\} \rangle$. Hence, the levels and potencies are maintained, but the feature set F_0 is decreased by removing from F_1 the features that appear in C. According to condition 1 in Definition 8, $\{\text{Gateways}\} = \{\text{Gateways, simple, object}\} \setminus (\{\text{object}\} \cup \{\text{simple}\})$. According to condition 2 in the definition, the formula Φ_0 is equivalent to replacing object by *true* and simple by *false* in Φ_1.
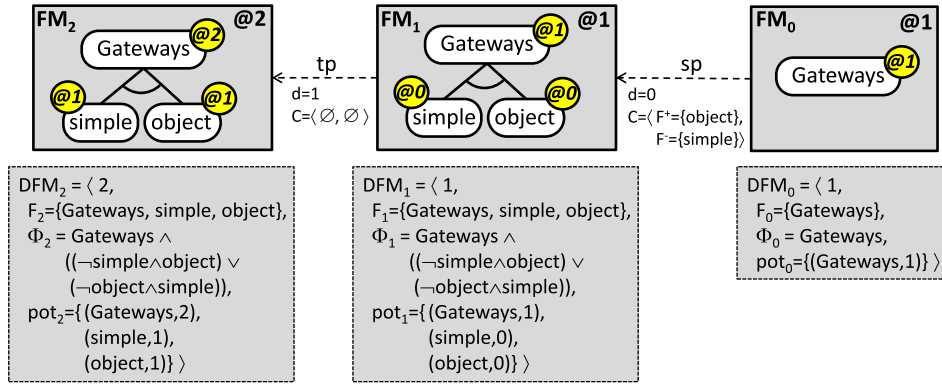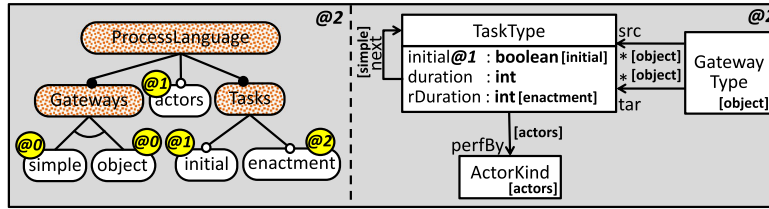
Fig. 9. Examples of F-morphisms.

Fig. 10. Deep model PL example.

F-morphisms are composable by adding their depths and constructing the union of the positive (resp. negative) features in the configurations, as the next definition and lemma show. Composition of F-morphisms is necessary as a basis for the results of Sect. 5.1.

Def. 9 (F-morphism composition). Given two F-morphisms $m_1 = \langle d^1, m_F^1, C^1 = \langle F_1^+, F_1^- \rangle\rangle\colon DFM_0 \to DFM_1$ and $m_2 = \langle d^2, m_F^2, C^2 = \langle F_2^+, F_2^- \rangle\rangle\colon DFM_1 \to DFM_2$, the composed morphism $m_2 \circ m_1\colon DFM_0 \to DFM_2$ is defined as $\langle d^1 + d^2, m_F^2 \circ m_F^1, \langle m_F^2(F_1^+) \cup F_2^+, m_F^2(F_1^-) \cup F_2^- \rangle\rangle$.

Lemma 2 (F-morphism composition yields an F-morphism). Given two F-morphisms $m_1 = \langle d^1, m_F^1, C^1 \rangle\colon DFM_0 \to DFM_1$ and $m_2 = \langle d^2, m_F^2, C^2 \rangle\colon DFM_1 \to DFM_2$, their composition $m_2 \circ m_1\colon DFM_0 \to DFM_2$ is a valid F-morphism.

Proof. By checking that the composition satisfies the five requisites for F-morphisms in Definition 8: correct depth, injectivity, composed configuration being correct, and conditions 1 and 2. See proof details in "Appendix". □

Example In Fig. 9, the composition of *sp* with *tp* results in the F-morphism $tp \circ sp$, which has depth 1 and configuration $C = \langle F^+ = \{object\}, F^- = \{simple\}\rangle$. This F-morphism models the combined action of instantiation and variability specialization, but is neither a type nor a specialization according to Definition 8.

Finally, we are ready to characterize deep model product lines (PLs) as a deep model, a deep feature model with the same level as the deep model, and a mapping assigning presence conditions (PCs) to deep model elements.

Def. 10 (Deep model PL). A deep model PL $DM = \langle M, DFM, \phi \rangle$ is made of:

- A deep model M and a deep feature model DFM with the same level ($p = l$).
- A function $\phi\colon C \cup S \cup R \to \mathbb{B}(F)$ mapping each element in M to a non-false propositional formula over the features in F, called *presence condition* (PC), s.t.:

  1. $\forall s \in S \cup R \bullet \phi(s) \Rightarrow \phi(src(s))$
  2. $\forall r \in R \bullet \phi(r) \Rightarrow \phi(tar(r))$
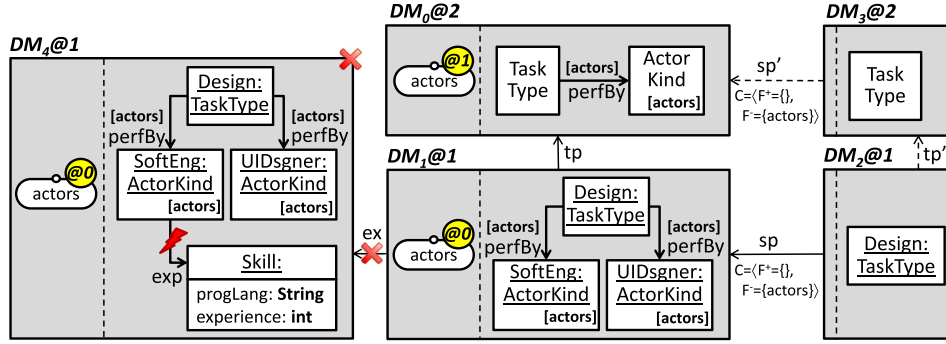 3. $\forall e \in C \cup S \cup R, \forall v \in Var(\phi(e)) \bullet pot(v) \le pot(e)$

Fig. 11. Examples of PL-morphisms and deferred configuration.

In the previous definition, we use function *Var* to return all variables (i.e., all features) within a propositional formula. Intuitively, given a configuration, we can derive a *product* (a deep model) of the PL by deleting the model elements whose PC evaluates to false when substituting its variables by their value *true* or *false*. To avoid dangling edges in product deep models, Definition 10 requires the PC of slots and references not to be weaker than the PC of their owning clabject (condition 1), and the PC of references not to be weaker than the one of their target clabject (condition 2). In addition, the variability of an element must be resolved in a level that contains the element or an instance of it. To this aim, condition 3 requires the potency of an element not to be smaller than the potency of the variables within its PC.

Example Figure 10 shows a deep model PL for process modelling languages. The left compartment contains the deep feature model, the one to the right contains the deep model, and the PCs are represented between square brackets close to the deep model elements they are mapped into. If an element does not specify a PC (like TaskType), then its PC is assumed to be *true*. This deep model PL permits two alternative realizations for gateways: either as the reference next if the feature simple is selected, or as the clabject GatewayType if the feature object is selected instead. This variability needs to be resolved before instantiating the language for a specific domain, as features simple and object have potency 0. The PL also offers the choice to add or not the primitive ActorKind to the language by means of the feature actors; since this feature has potency 1, this decision can be taken either before specializing the language or at level 1 to enable exploratory modelling. Finally, the PL allows selecting whether tasks can be initial or hold enactment information. By condition 3 in Definition 10, feature initial in the feature model cannot have potency 2 because the feature is used in the PC of attribute TaskType.initial, which has potency 1. The feature model depicts features ProcessLanguage, Gateways and Tasks in colour and without a potency; this is so as these features are mandatory (i.e., *true* in any valid configuration), and while the figure shows them to obtain a tree-like feature model, the formalization of the example does not include them.

Next, we introduce mappings between deep model PLs (called PL-morphisms) as a tuple of morphisms between their constituent deep models and deep feature models. As in the previous cases, we are interested in type morphisms, linguistic extensions, and specializations of deep model PLs via a (partial) configuration.

Def. 11 (PL-morphism, type, extension, specialization). Given two deep model PLs $DM_i = \langle M_i, DFM_i, \phi_i \rangle$ (for $i = \{0, 1\}$), a PL-morphism $m = \langle m^D, m^F \rangle$ consists of a D-morphism $m^D \colon M_0 \to M_1$ and an F-morphism $m^F \colon DFM_0 \to DFM_1$ with configuration $C = \langle F_1^+, F_1^- \rangle$, s.t.:

$$\forall\, e \in C_0 \cup S_0 \cup R_0 \;\bullet\; \phi_1(m^D(e))[F_1^+/true, F_1^-/false] \cong \phi_0(e)[F_0/m_F^F(F_0)]$$

PL-morphism $tp = \langle tp^D, tp^F \rangle$ is a *type* if both tp^D and tp^F are types.
PL-morphism $ex = \langle ex^D, id^F \rangle$ is an *extension* if ex^D is an extension and id^F is an identity.
PL-morphism $sp = \langle m^D, sp^F \rangle$ is a *specialization* if sp^F is a specialization and m^D is injective, level-preserving, and satisfying $\forall\, e \in C_1 \cup S_1 \cup R_1 \;\bullet\; (\phi_1(e)[F_1^+/true, F_1^-/false] \not\cong false \Leftrightarrow \exists\, e' \in C_0 \cup S_0 \cup R_0 \;\bullet\; m^D(e') = e)$.

Remark There is no condition on the equality of depths of m^D and m^F, as M_0 and DFM_0 have the same level, and in turn, the levels of M_1 and DFM_1 are equal. The condition for PL-morphisms demands that the PCs in the deep model M_0 are modified according to the selection of features in configuration C of m^F. In addition, in specialization PL-morphisms, M_0 should contain just the elements whose PC is not false after substituting the features in F^+ by *true*, and the ones in F^- by *false*. Therefore, the definition of specialization PL-morphism requires that only the elements in M_1 whose PC is not false after substituting the features by their values, are

mapped from M_0; while this mapping m^D needs to be injective. Moreover, by Definition 10 of deep model PL, no element in M_0 can have a PC that is false.

Other kinds of PL-morphisms are possible, for example, adding features to a feature model in the same or lower levels to increase its variability. We will introduce this possibility in Sect. 5.2.

Example Figure 11 shows four PL-morphisms (tp, tp', sp, sp') and a function ex, which fails to be a PL-morphism. Both tp and tp' are types: they relate models at adjacent levels, where one is an instance of (typed by) the other. Types always have depth 1 and use the empty configuration $C = \langle \emptyset, \emptyset \rangle$ (cf. Definition 8), and so, a model element and its instances have the same PC (see, e.g., ActorKind and its instances SoftEng and UIDsgner). Both sp and sp' are specialization PL-morphisms. This is so as they preserve levels and potencies, and map injectively the model elements whose PC does not evaluate to *false* when applying the configuration C. Actually, since the configuration C of both PL-morphisms is total, the PC of the elements in DM_3 and DM_2 evaluates to true, and hence, these models do not have more closed variability options to configure (i.e., they are final products of the PL).

The figure also shows a deep model DM_4, which is an attempt to extend DM_1 by a linguistic extension made of the clabject Skill and its incoming reference exp. However, the result is not a valid deep model PL as the PC of exp ($true$) is weaker than the PC of its owner clabject SoftEng (actors). This makes ex fail to be a PL-morphism. DM_4 can become a deep model PL by adding to exp and Skill the PC actors. In such a case, morphism ex (with empty configuration) would become a PL-morphism.

Finally, we define PL-morphism composition, and show that it leads to a valid PL-morphism.

Def. 12 (PL-morphism composition). Given PL-morphisms $m_1 = \langle m_1^D, m_1^F \rangle \colon DM_0 \to DM_1$ and $m_2 = \langle m_2^D, m_2^F \rangle$ $\colon DM_1 \to DM_2$, the composition morphism $m_2 \circ m_1 \colon DM_0 \to DM_2$ is defined as $\langle m_2^D \circ m_1^D, m_2^F \circ m_1^F \rangle$.

Lemma 3 (PL-morphism composition yields a PL-morphism). Given PL-morphisms $m_1 = \langle m_1^D, m_1^F \rangle \colon DM_0 \to DM_1$ and $m_2 = \langle m_2^D, m_2^F \rangle \colon DM_1 \to DM_2$, their composition $m_2 \circ m_1 \colon DM_0 \to DM_2$ is a valid PL-morphism. If m_1 and m_2 are specializations, so is $m_2 \circ m_1$.

Proof. For proving the first part of the lemma, we use Lemmas 1 and 2 (composition of D- and F-morphisms), and then check the condition in Definition 11. For the second part, we check that the resulting morphism is injective, level-preserving and that the co-domain keeps only the elements with non-false PC. See proof details in "Appendix". □

5. Engineering and using language families via multi-level model product lines

In this section, we apply and extend the theory presented so far to cover two scenarios in language family engineering. The first one (Sect. 5.1) is its usage to represent a language family with variability that is specialized via instantiation and (partial) configurations. The second one (Sect. 5.2) covers the creation process of a language family with variability, which can be done either top-down (by working at the top-level to incrementally extend the language variability) or bottom-up (by pulling up linguistic extensions and variability options to upper levels).

5.1. Usage of a language family

A deep model PL can be used as a language family with variability. Figure 12 illustrates this usage scenario. The deep model with variability at the top describes a language family. This language is created by a language family designer in step 1. While the figure depicts a language family with depth 3, our framework is general and supports any depth.

Then, the figure depicts two scenarios to the left and right. In the branch (a) to the right, a DSL designer (labelled DSL-1 designer) customizes the language family by giving a total configuration (step 2a). This resolves all "closed" variability options offered by the language family definition before using the language. Then, the DSL designer instantiates the language to create the DSL-1 meta-model (step 3a), which DSL-1 users can instantiate to build models (step 4a). Compared with a standard software language engineering process, here the DSL designer uses a language family meta-model as a basis to create DSL-1, instead of using a meta-modelling language like the OMG's Meta-Object Facility [MOF16]. The advantage is that the language family meta-model contains relevant primitives for the DSL scope (e.g., TaskType, ActorKind), which do not need to be invented anew, but specialized for the domain via instantiation. Moreover, the language family meta-model can define services like transformations or code generators, which can be reused for every DSL of the family [dLGC15].
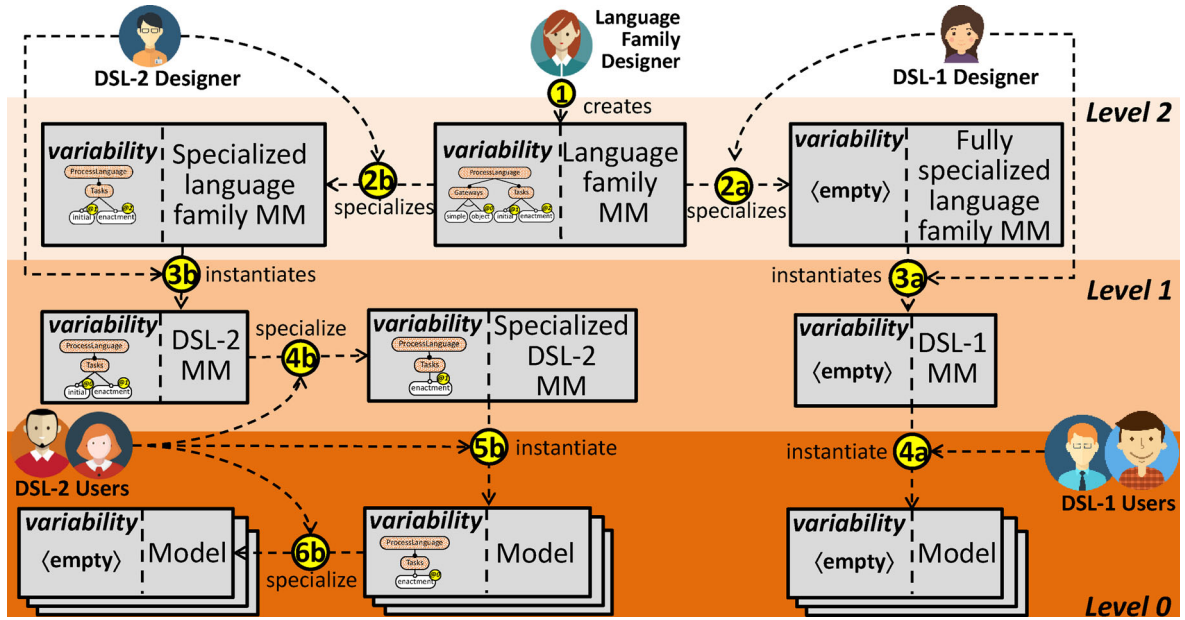
Fig. 12. Using a language family with variability. Branch (a—right) resolves the closed variability options before language use. Branch (b—left) defers variability resolution to lower levels.

The branch (b) to the left illustrates a more flexible usage scenario, where some "closed" variability options are not resolved at the top level, but later at lower levels, including level 0. In this case, a DSL designer (labelled DSL-2 designer) specializes the language family definition by providing a partial configuration (step 2b), and then instantiates the resulting language to define the DSL-2 meta-model (step 3b). However, in this case, some variability options remain open at level 1. Hence, the DSL users can specialize the language further according to their needs (step 4b), and then use it to create models (step 5b). In the figure, some variability remains open at level 0, meaning that the DSL users can create models with variability and resolve this variability later (step 6b).

We need to tackle two issues to properly realize both scenarios. The first one relates to *derivation*. Since our theory models configurations by means of specialization morphisms, the question is whether for any (total or partial) configuration, we can find a corresponding valid specialization morphism. This enables steps 2a, 2b, 4b, and 6b in Fig. 12. The second one pertains scenario (b). Since we want to allow deferring the variability resolution to lower levels, the question is whether there is always an equivalent scenario (a) where all the variability is resolved in a first step. If so, this entails that the expressiveness of a language family with variability is independent on how it is used.

We first tackle the issue concerning derivation. When the configuration C of a specialization PL-morphism $sp\colon DM_0 \to DM_1$ is total, DM_0 is a *product* of DM_1 with no variability options to choose from, being equivalent to a deep model (without the PL part, cf. Definition 1). This is so as the feature model would be empty, and all PCs of the model elements would be *true*. However, the question remains whether for any valid configuration C of a deep model PL DM, we can find a deep model PL DM' and a specialization PL-morphism $sp\colon DM' \to DM$ via C. This requires showing that any choice of feature configuration $\langle F^+, F^- \rangle$ produces a valid deep model PL DM' as given by Definition 10. Theorem 5.1 captures this result.

Theorem 5.1 *(Specialization morphisms for configurations)* Given a deep model PL $DM = \langle M, DFM, \phi \rangle$ and any configuration C of DFM, there is exactly one deep model PL DM', such that a specialization PL-morphism $sp\colon DM' \to DM$ with configuration C exists.

Proof. We construct a deep model PL DM', where its model M' has same level as M, and contains the elements of M with non-false PC. Similarly, the feature model of DM' is restricted to the features that have not been set by C (i.e., those remaining undefined) and the formula is the result of evaluating DFM's formula on configuration C. Then, we prove that such deep model PL is valid according to Definition 10. Finally, we build a specialization PL-morphism from DM' to DM, showing that it fulfils Definition 11 and is unique. See proof details in the "Appendix". □

$$DM_0 \overset{sp'}{\dashleftarrow} DM_3$$

$$\uparrow tp \qquad = \qquad \uparrow tp'$$

$$DM_1 \overset{sp}{\leftarrow} DM_2$$

Fig. 13. Deferred configuration: specialization can be advanced to instantiation.

We are now ready to characterize the process of derivation from a deep model PL via a configuration, using specialization PL-morphisms and the results of Theorem 5.1.

Def. 13 (Derivation). Given a deep model PL $DM = \langle M, DFM = \langle l, FM, pot \rangle, \phi \rangle$, the set of derivable deep model PLs $Der(DM) = \{DM' \mid \exists sp : DM' \to DM \text{ with configuration } C \in CFG(FM)\}$ is made of the set of all deep models DM' s.t. there is a specialization PL-morphism $sp : DM' \to DM$ using any (total, partial) configuration C of DFM. DM' is called a (total, partial) product of DM.

Next, we look into the second issue, which is the soundness of deferring the configuration of an element after it is instantiated. The question is whether, in every situation that allows configuring an element after its instantiation, we obtain the same result by resolving the element variability first and then instantiating. This result is important as, regardless of the order in which configurations and instantiation are performed, we can calculate the language that results from applying the configurations as the first step, by advancing the configuration steps over the instantiations.

The next theorem captures the fact that if we can instantiate and then configure, then we obtain the same result if we configure and then instantiate.

Theorem 5.2 (Specialization can be advanced to instantiation). Given three deep model PLs $DM_i = \langle M_i, DFM_i, \phi_i \rangle$ (for $i = \{0, 1, 2\}$), a type PL-morphism $tp : DM_1 \to DM_0$ and a specialization PL-morphism $sp : DM_2 \to DM_1$, there is a unique deep model PL $DM_3 \in Der(DM_0)$, a specialization PL-morphism $sp' : DM_3 \to DM_0$ and a type PL-morphism $tp' : DM_2 \to DM_3$ s.t. the diagram in Fig. 13 commutes.

Proof. We use Theorem 5.1 to construct a deep model PL DM_3 and a specialization morphism $sp' : DM_3 \to DM_0$, using the configuration of specialization morphism sp. Then, a well-defined, unique type PL-morphism from DM_2 to DM_3 can be constructed by restricting tp. See proof details in "Appendix". □

Remark The converse is not true in general, that is, instantiation cannot always be advanced to specialization. The reason is that type morphisms to features with potency 0 are disallowed, and so they must be configured first.

Example Figure 11 shows a deferred configuration. Deep model PL DM_0 is instantiated into DM_1, and then specialized using the configuration $C = \langle F^+ = \{\}, F^- = \{actors\} \rangle$ to yield DM_2. As the figure shows, we obtain the same result by first specializing DM_0 using C, which yields DM_3, and then instantiating DM_3 into DM_2. Deep model PL DM_3 is relevant since it corresponds to the fully-configured language (i.e., with no unresolved variability) employed to build DM_2.

Finally, we use Theorem 5.3 to ensure that, given any arbitrary chain of specializations (via partial configurations) and instantiations, we can calculate the fully configured language by applying all configurations in one step. This allows going from scenario (b) to scenario (a) in Fig. 12 by iterating the construction of Theorem 5.2 (cf. Fig. 13).

Theorem 5.3 (Equivalent fully configured language). Given a chain of PL-morphisms:

$$DM_0 \overset{tp_1}{\leftarrow} DM_1 \overset{sp_2}{\leftarrow} DM_2 ... \overset{tp_n}{\leftarrow} DM_n \overset{sp_{n+1}}{\leftarrow} DM_{n+1}$$

where each tp_i is a unique type PL-morphism and each sp_i is a specialization PL-morphism, there is:

1. a unique deep model PL $DM_{FC} \in Der(DM_0)$, called *fully configured language*, and
2. a specialization PL-morphism $sp'_0 : DM_{FC} \to DM_0$ with a configuration that selects and discards the same features as the configurations of $sp_2, ..., sp_{n+1}$,

such that DM_{n+1} is an (indirect) instance of DM_{FC}.

Proof. This proof uses Theorem 5.2 to advance specialization to type PL-morphisms, and the fact that PL-morphisms can be composed. See proof details in "Appendix". □
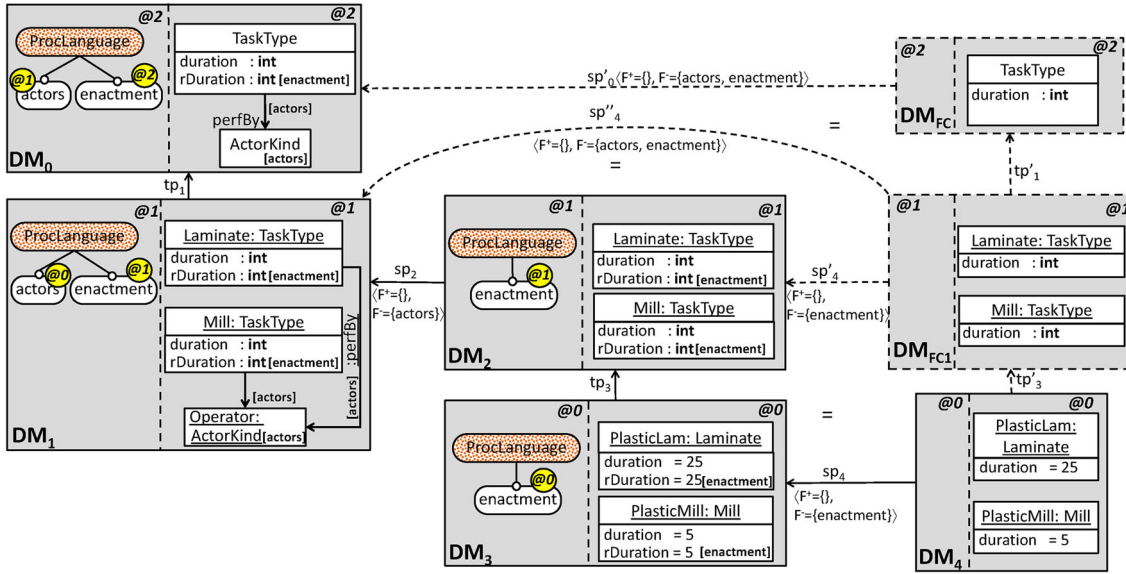
Fig. 14. Calculating the equivalent fully configured language of the exploratory modelling chain $DM_0 \leftarrow DM_1 \leftarrow DM_2 \leftarrow DM_3 \leftarrow DM_4$.
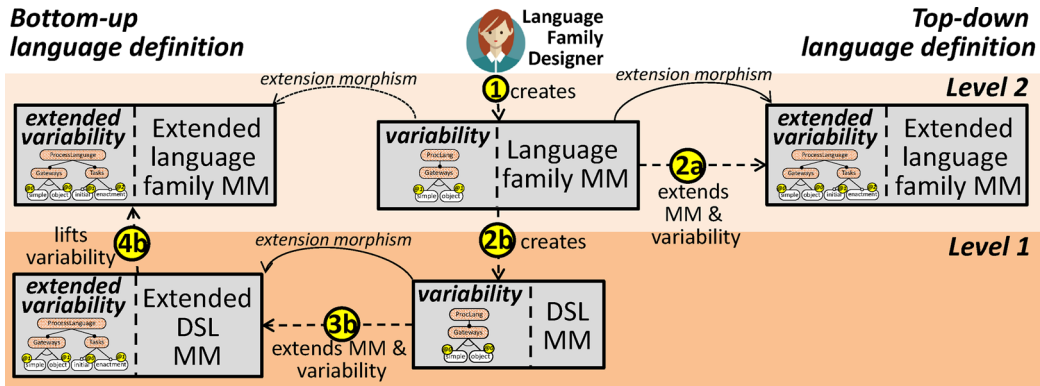

Fig. 15. Creating a language family with variability. **a** Top-down approach. **b** Bottom-up approach.

Example Figure 14 shows an example of calculation of the equivalent fully configured language (DM_{FC}) of the chain $DM_0 \leftarrow DM_1 \leftarrow DM_2 \leftarrow DM_3 \leftarrow DM_4$. The diagram depicts a scenario where the language family meta-model (DM_0) is instantiated for the industrial process domain (DM_1), and two instances of TaskType (Laminate and Mill) and one of ActorKind (Operator) are defined. Subsequently, the designer decides to discard ActorKinds from the language since both task types are automated. This is done by the specialization morphism sp_2, where the configuration sets feature actors to *false*, and the result is DM_2. Then, this industrial process model language description DM_2 is instantiated into DM_3. At this point, the modeller realizes that the real task duration (rDuration attribute) is always the same as the expected task duration (duration attribute), since tasks are automated. Hence, a specialization sp_4 that sets feature enactment to *false* is performed, yielding model DM_4. DM_4 does not have any variability options to configure.

Now, we use Theorem 5.3 to calculate the equivalent fully configured language (DM_{FC}) in the figure. This is the indirect deep model type that results from doing all the specializations before any instantiation, and corresponds to the language definition without variability needed to create DM_4. First, we use Theorem 5.2 to build DM_{FC1}. This is the deep model type of DM_4 without variability options at level 1. Then, we use the composition morphism $sp_2 \circ sp'_4$ to iterate the same construction to yield DM_{FC}. Finally, we compose $tp'_1 \circ tp'_3$ to obtain an indirect type morphism $DM_4 \rightarrow DM_{FC}$.

For clarity, note that models DM_1, DM_2 and DM_{FC1} explicitly show the attributes rDuration and/or duration, but these are normally hidden in practical tools.
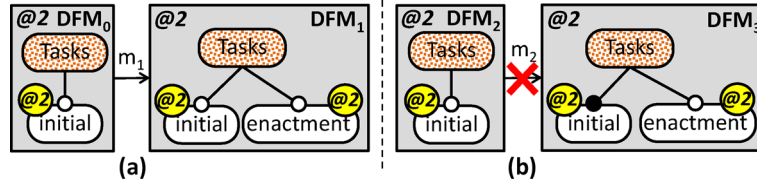
Fig. 16. Correct (**a**) and incorrect (**b**) EF-morphisms.

5.2. Engineering a language family with open and closed variability

In this subsection, we turn our attention at how language families are constructed with our approach, based on the notion of *extension*. As it is common in standard practice of SPLs [KC16], we anticipate two processes called *top-down* and *bottom-up*, as depicted in Fig. 15.

In the *top-down* approach, shown in branch (a) to the right of the figure, the variability options are defined up-front together with the language meta-model. This requires support for extension to increase both the feature model and the meta-model of the language family (step with label 2a). For this purpose, we will define a new kind of morphism between deep model PLs to represent such extensions.

In the *bottom-up* approach, shown in branch (b) to the left of the figure, concrete language instantiations at low levels guide the creation or extension of the language family definition at the top level. For example, the language family designer may instantiate a draft version of the language family meta-model for exploratory modelling (step 2b), and then realize that the given domain requires adding linguistic extensions to the instance or extending the feature model with additional variability options (step 3b). If such extensions are deemed general, a lifting process can promote them one level up, in the figure from level 1 to level 2 (step 4b). This way, the original language family definition becomes extended. Moreover, if the designer specializes the language family via a configuration before performing the extension at level 1, then we need to advance the extension to the specialization to incorporate the new variability to the top level. For this purpose, we will have to characterize compatibility conditions between extension and specialization. While the proposed bottom-up techniques may enable the creation of a language family from scratch out of a set of existing individual sample languages, we will tackle this scenario in future work.

In the remainder of this section, we first introduce extension morphisms to enable increasing the variability of deep model PLs. Then, we define mechanisms to advance extension to specialization and to instantiation in order to enable the bottom-up construction of deep model PLs. We start by defining extension morphisms (EF-morphisms) between deep feature models. Since we aim for interleaving specializations and extensions, we allow feature model extensions only if they preserve all (partial) configurations of the feature model. Definition 14 formalizes this intuition.

Def. 14 (EF-morphism). Given two deep feature models $DFM_i = \langle l_i, FM_i = \langle F_i, \Phi_i \rangle, pot_i \rangle$ (for $i = \{0, 1\}$) with the same level, a variability extension deep feature model morphism (EF-morphism in short), written $DFM_0 \stackrel{m_e}{\Longrightarrow} DFM_1$, is an injective set morphism $m_e \colon F_0 \to F_1$ such that:

1. $\forall f \in F_0 \ \bullet \ pot_0(f) = pot_1(m_e(f))$
2. $\forall \ C_0 = \langle F_0^+, F_0^- \rangle \in CFG(FM_0), \ \exists \ C_1 = \langle F_1^+, F_1^- \rangle \in CFG(FM_1) \ \bullet \ m_e(F_0^+) \subseteq F_1^+ \wedge m_e(F_0^-) \subseteq F_1^- \wedge$
   $\forall \ C_0 = \langle F_0^+, F_0^- \rangle \notin CFG(FM_0), \ \nexists \ C_1 = \langle F_1^+, F_1^- \rangle \in CFG(FM_1) \ \bullet \ m_e(F_0^+) \subseteq F_1^+ \wedge m_e(F_0^-) \subseteq F_1^-$

Condition 1 in the definition preserves the potencies. Condition 2 ensures that any (partial) configuration $\langle F_0^+, F_0^- \rangle$ of FM_0 can be extended to a (partial) configuration of FM_1 by expanding the sets F_0^+ and F_0^-. Hence, modulo feature renaming by m_e, we have $C_0 \leq C_1$. In addition, condition 2 prevents extending invalid configurations of FM_0 to valid configurations of FM_1.

Example Figure 16 shows two examples of allowed and disallowed extensions of deep feature models. In Fig. 16a the deep feature model DFM_0 is extended with a new optional feature called enactment. Morphism m_1 is a valid extension, since any non-valid configuration in DFM_0 cannot be extended to a valid one in DFM_1, and any valid configuration in DFM_0 can be extended to a valid one in DFM_1. However, morphism m_2 in Fig. 16b is not a valid EF-morphism. In this case, the extension attempt adds an optional feature called enactment and makes the existing feature initial mandatory. This is not a valid EF-morphism since configurations of DFM_2 making initial false ($\langle F^+ = \{\}, F^- = \{initial\} \rangle$) are valid, but cannot be extended to valid configurations of DFM_3.
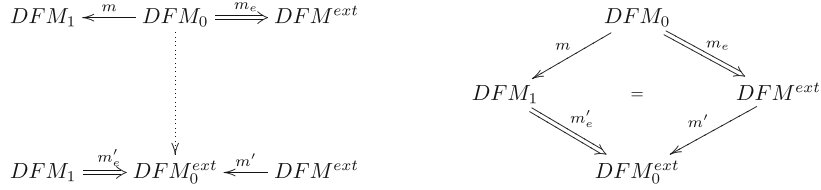
$$DFM_1 \xleftarrow{\;m\;} DFM_0 \xRightarrow{\;m_e\;} DFM^{ext}$$

$$DFM_1 \xRightarrow{\;m'_e\;} DFM_0^{ext} \xleftarrow{\;m'\;} DFM^{ext}$$

(right commuting square diagram)

Fig. 17. Advancing extension to specialization/typing (left). Commuting square (right).
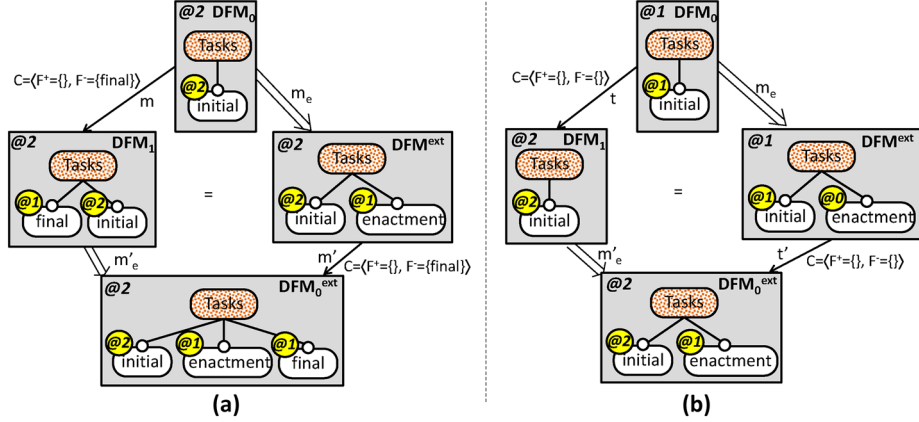

(a)

(b)

Fig. 18. Examples of advancing EF-morphisms to F-morphisms. **a** Advancing through a specialization F-morphism. **b** Advancing through a type F-morphism.

Remark We cannot use the F-morphisms from Definition 8 to represent extensions, since F-morphisms include a (partial) configuration modelling a specialization. For example, in the EF-morphism m_1 in Fig. 16a, this implies being able to assign a *true* or *false* value to feature enactment, which is not possible because the feature does not exist in DFM_0. Moreover, condition 1 in Definition 8 of F-morphisms requires DFM_0 and DFM_1 to have the same feature set, and condition 2 requires they have equivalent formula modulo their partial evaluation using the morphism configuration.

Next, we need to check that EF-morphisms can be advanced to specialization (to enable advancing variability extensions to configurations) and to type F-morphisms (to allow lifting variability extensions to upper levels). Lemma 4 formalizes this.

Lemma 4 (EF-morphisms can be advanced to F-morphisms). Given an F-morphism $m = \langle d, m_F, C \rangle \colon DFM_0 \to DFM_1$ and an EF-morphism $DFM_0 \xRightarrow{\;m_e\;} DFM^{ext}$, there is a deep feature model DFM_0^{ext}, an F-morphism $m' = \langle d, m'_F, C \rangle \colon DFM^{ext} \to DFM_0^{ext}$, and an EF-morphism $DFM_1 \xRightarrow{\;m'_e\;} DFM_0^{ext}$ s.t. $m'_e \circ m_F = m'_F \circ m_e$ as Fig. 17 shows.

Proof. We first construct the deep feature model DFM_0^{ext} by the union of features of DFM_1 and DFM^{ext}, and the disjunction of their formulae. Then, morphisms m' and m'_e are built and shown to commute. See proof details in "Appendix". $\square$

Example Figure 18a shows an example of advancing an EF-morphism (m_e in the figure) to a specialization F-morphism (m in the figure). F-morphism m models a specialization of DFM_1 that assigns *false* to feature final. The EF-morphism m_e models the addition of an optional feature enactment to DFM_0. In this setting, we can construct a deep feature model DFM_0^{ext} by merging DFM_1 and DFM^{ext}. We obtain an extension EF-morphism m'_e which models the addition of feature enactment to DFM_1, and a specialization morphism m' that assigns the value *false* to feature final. Overall, we have been able to advance the extension over the configuration: in $DFM_1 \xleftarrow{\;m\;} DFM_0 \xRightarrow{\;m_e\;} DFM^{ext}$, we configure DFM_1 and then extend the variability; while in $DFM_1 \xRightarrow{\;m'_e\;} DFM_0^{ext} \xleftarrow{\;m'\;} DFM^{ext}$, we first extend DFM_1 and then configure.

The advancement of extensions through type F-morphisms is also possible. Figure 18b shows an example, where the potency of feature enactment in DFM_0^{ext} is obtained by adding the depth of morphism t to the potency of the feature in DFM^{ext}.
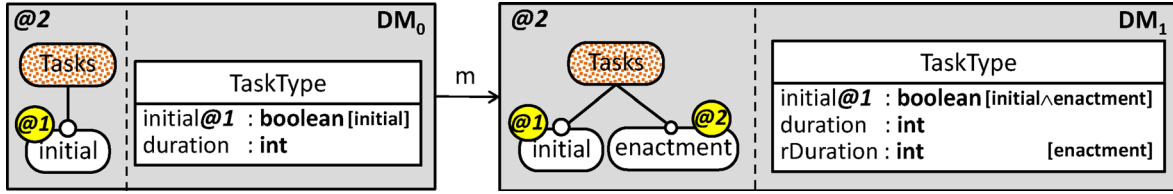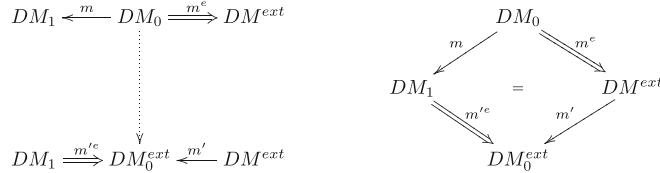
Fig. 19. Example of EPL-morphism.

Fig. 20. Advancing extension to specialization/typing (left). Commuting square (right).

Remark Technically, DFM_0^{ext} is not unique because there are many equivalent formulae Φ modelling the same feature model.

Now, we use EF-morphisms to define extensions of deep model PLs. For this purpose, we define EPL-morphisms. Unlike PL-morphisms in Definition 11, EPL-morphisms permit making the PC of elements stronger, which is useful to allow the PC of existing elements to use the new features added by a deep feature model extension. Definition 15 captures this new type of morphism.

Def. 15 (EPL-morphism). Given two deep model PLs $DM_i = \langle M_i, DFM_i, \phi_i \rangle$ (for $i = \{0, 1\}$) with same level, an extension PL-morphism (EPL-morphism in short) $m = \langle m^D, m_e \rangle$, written $DM_0 \stackrel{m}{\Longrightarrow} DM_1$, is made of a D-morphism $m^D \colon M_0 \to M_1$ and an EF-morphism $DFM_0 \stackrel{m_e}{\Longrightarrow} DFM_1$ s.t. $\forall e \in C_0 \cup S_0 \cup R_0 \bullet \phi_1(m^D(e)) \Rightarrow \phi_0(e)[F_0/m_e(F_0)]$.

Example Figure 19 shows an example EPL-morphism m. On the one hand, its constituent EF-morphism expands the feature model with an additional optional feature enactment. On the other, its D-morphism adds a new attribute rDuration to clabject TaskType. The PC of attribute initial is refined to initial∧enactment, which the condition in Definition 15 allows since initial∧enactment ⇒ initial. The new attribute rDuration is assigned a PC, which is also allowed since the attribute is not mapped from DM_0.

Remark We cannot use standard PL-morphisms to represent EPL-morphisms, as only the former but not the latter include a configuration. In the example of Fig. 19, the configuration could assign the value *false* to enactment to emulate that DM_0 does not define attribute rDuration, but this attribute could define any PC (e.g., enactment∨initial) that may not evaluate to *false*. Moreover, the PC of initial in DFM_1 is initial∧enactment, which does not evaluate to initial when substituting enactment by *false*.

We are ready to enunciate the main results of this subsection. First, Theorem 5.4 states that EPL-morphisms can be advanced to (both type and specialization) PL-morphisms under certain conditions. Then, Corollary 1 states that these conditions are enough for type PL-morphisms, while specialization PL-morphisms require an additional condition guaranteeing the compatibility of the extension with the configuration chosen for the specialization.

Theorem 5.4 *(EPL-morphisms can be advanced to PL-morphisms)* Let $m = \langle m^D, m^F \rangle \colon DM_0 \to DM_1$ be a PL-morphism and $m^e = \langle m_e^D, m_e \rangle \colon DM_0 \to DM^{ext}$ an EPL-morphism, such that:

1. $\forall e_i, e_j \in C_0 \cup S_0 \cup R_0 \bullet m^D(e_i) = m^D(e_j) \Rightarrow \phi^{ext}(m_e^D(e_i)) = \phi^{ext}(m_e^D(e_j))$
2. $\forall e \in C_0 \bullet \phi^{ext}(m_e^D(e)) = \phi_0(e)[F_0/m_e(F_0)]$.

We can build a deep model PL DM_0^{ext}, a PL-morphism $m' = \langle m'^D, m'^F \rangle \colon DM^{ext} \to DM_0^{ext}$ and an EPL-morphism $DM_1 \stackrel{m'e}{\Longrightarrow} DM_0^{ext}$ with $m'^e = \langle m_e'^D, m_e' \rangle$ s.t. $m_e' \circ m^F = m'^F \circ m_e$ and $m_e'^D \circ m^D = m'^D \circ m_e^D$ as Fig. 20 shows.
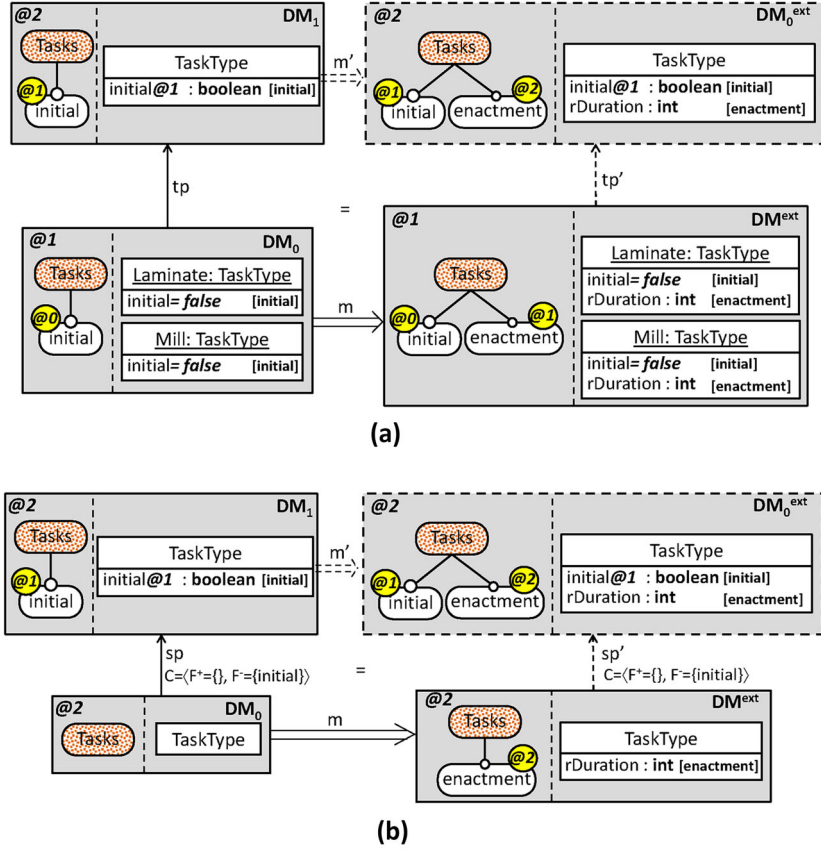
Fig. 21. Examples of advancing EPL-morphisms to PL-morphisms. **a** Advancing through a type PL-morphism. **b** Advancing through a specialization PL-morphism.

Proof. We first construct the deep model PL DM_0^{ext}, where the model part is built by a construction similar to a pushout in graphs, and the deep feature model is built as in the proof of Lemma 4. Then, the morphisms m' and m'_e are constructed and shown to commute. See proof details in "Appendix". $\square$

In the previous theorem, condition 1 requires that the elements in DM_0 that are mapped to the same element in DM_1 have the same PC in DM^{ext}. In addition, condition 2 forbids that the extension modifies the PC of clabjects to avoid dangling references and slots when advancing the extension morphism.

Corollary 1 (Preservation of type and specialization PL-morphisms). Let $m = \langle m^D, m^F \rangle \colon DM_0 \to DM_1$ be a PL-morphism and $m^e = \langle m_e^D, m_e^F \rangle \colon DM_0 \to DM^{ext}$ an EPL-morphism satisfying the conditions of Theorem 5.4, and let m' be the advanced PL-morphism. Then:

1. If m is a type PL-morphism, so is m'.
2. If m is a specialization, and $\forall e \in C^{ext} \cup S^{ext} \cup R^{ext} \bullet \phi^{ext}(e)[F^+/true, F^-/false] \not\cong false$, then m' is a specialization.

Proof. Regarding 1) by construction m' and m have same depth, and so if m is type, so is m'. For 2) in addition, we need to check injectivity, and the condition on the PCs as per Definition 11. See details in "Appendix". $\square$

Example Figure 21 illustrates the advancement of EPL-morphisms to PL-morphisms. In particular, Fig. 21a shows an example of extension that is advanced to instantiation. We start from a situation where DM_1 has been instantiated into DM_0, which has two instances of TaskType, and next, DM_0 has been extended into DM^{ext} by adding a new feature enactment and a linguistic extension rDuration to both task types. Then, by using Theorem 5.4, we build model DM_0^{ext} and morphisms tp' and m'. This means that we can first extend DM_1 (via m') to yield DM_0^{ext} and then instantiate DM_0^{ext} (via tp') to yield DM^{ext}. However, not any DM^{ext} permits this advancement,

as the condition of Theorem 5.4 states. For example, should we assign the PC enactment∧initial to Laminate.rDuration, we would not obtain a correct type PL-morphism m'. This is so as Laminate.rDuration and Mill.rDuration are both mapped to the same clabject TaskType.rDuration in DM_0^{ext}, but they would have different PCs.

Figure 21b shows that Theorem 5.4 can also be applied to advancing extension to specialization. In the figure, we start from a situation where DM_1 has been specialized via the PL-morphism sp with configuration $C = \langle F^+ = \{\}, F^- = \{initial\}\rangle$ to yield DM_0, and then, DM_0 has been extended via the EPL-morphism m to yield DM^{ext}. DM^{ext} contains an extra feature enactment and a linguistic extension rDuration. We can use Theorem 5.4 to obtain DM_0^{ext} and morphisms m' and sp'. This way, we can first extend DM_1 (via m') and then specialize it (via sp'). The PC of rDuration in DM^{ext} satisfies the condition of Corollary 1, since enactment[initial/false] = enactment $\not\cong$ false. Should the PC be enactment∧initial, then sp' would not be a proper specialization. This is so as enactment∧initial[initial/false] = false, and so the field TaskType.rDuration should not be present in DM^{ext}. In this case, the extension would collide with the specialization, precluding the advancement.

6. Tool support

We have implemented the notions presented so far atop METADEPTH [dLG10]. This is a textual multi-level modelling tool which supports an arbitrary number of meta-levels and deep characterization through potency. It integrates the Epsilon family of languages for model management [PKR+09], which permits defining code generators and model transformations for multi-level models.

METADEPTH was used to define language families via multi-level modelling in [dLGC15], but it did not support the definition of closed sets of variability options by means of PLs. For this work, we have extended the tool to allow creating deep feature models and multi-level models with PCs, specializing deep model PLs via configurations, extending them with new features, and advancing those extensions to instantiations. The extended tool is available at http://metadepth.org/pls, together with examples of use.

In the following we showcase the use of the tool for three different activities: top-down creation of language families (Sect. 6.1), use of language families (Sect. 6.2) and bottom-up extension of language families (Sect. 6.3). Overall, these activities cover the scenarios explained in Sects. 5.1 and 5.2.

6.1. Top-down creation of language families

METADEPTH has a uniform textual syntax to specify models at any meta-level, similar to the UML Human-Usable Textual Notation [OMG04]. In addition, the tool supports the specification of domain-specific textual syntaxes for deep language families [dLGC15]. While the approach fosters a textual approach to modelling, the tool is able to produce read-only graphical views with the models being built, which may help in their comprehension.

Listing 1 specifies the deep model in the right part of Fig. 10, using METADEPTH's syntax. First, line 1 states the name of the deep feature model (defined in Listing 2) associated to the deep model. Then, line 2 declares a deep model with level 2, named ProcessModel. This contains three clabjects: TaskType (lines 3–13), ActorKind (lines 15–16) and GatewayType (lines 18–22). PCs are specified as @Presence annotations. This is possible as, similar to Java [CdL16], METADEPTH permits defining annotation types by providing their syntax, parameters, and the kind of elements they can annotate (models, clabjects or fields) [CdL18]. This definition is a meta-model, and so, when annotations are parsed, they are transformed into a model conforming to such meta-model. The model with the parsed annotations contains references to the annotated model (e.g., ProcessModel in this case). Representing annotations as models, allows well-formedness checking of the specific annotations with respect to their definition (i.e., the annotation values are not just uninterpreted strings).

```
1   @Variability(model="ProcessOptions")
2   Model ProcessModel@2 {
3     Node TaskType {
4       @Presence(condition="initial")
5       initial@1 : boolean = false;
6       duration : int;
7       @Presence(condition="enactment")
8       rDuration : int;
9       @Presence(condition="simple")
10      next : TaskType;
11      @Presence(condition="actors")
12      perfBy : ActorKind;
13    }
14
15    @Presence(condition="actors")
16    Node ActorKind;
17
18    @Presence(condition="object")
19    Node GatewayType {
20      src : TaskType[*];
21      tar : TaskType[*];
22    }
23  }
```

Listing 1: Deep model PL.

Regarding the PC of fields, for usability reasons, our implementation internally conjoins the PC of fields with the PC of their owner clabject. For example, the PC of reference Gateway.src is object, because the PC of Gateway is object. This guarantees that condition 1 in Definition 10 is satisfied, while conditions 2 and 3 are checked by constraints. Finally, please note that, while the condition parameter of the Presence annotation is a String, we internally check that it is a well formed boolean formula, which uses the features of the feature model identified in the Variability annotation of the model.

We have created a meta-model for deep feature models, and designed a domain-specific textual syntax for it, similar to the FAMILIAR tool [ACLF13]. Listing 2 shows the METADEPTH definition of the deep feature model in Fig. 10 (but we changed the potency of features simple and object to 1). Line 1 declares a feature model called ProcessOptions with level 2. Line 2 declares the root feature ProcessLanguage, and its children features Gateways, Tasks and actors. Children features can specify a potency after the "@" symbol, and be declared optional using the "?" symbol. Line 3 declares the children of Gateways, which are alternative as specified by the keyword alt. Line 4 declares the children of Tasks, which are optional.

```
1   FeatureModel ProcessOptions@2 {
2       ProcessLanguage : Gateways Tasks actors?@1;
3       alt Gateways : simple@1 object@1;
4       Tasks : initial?@1 enactment?@2;
5   }
```

Listing 2: Deep feature model.

Figure 22 shows the internal representation of a deep model PL in METADEPTH. The PC annotations are automatically converted into an annotation model, which is also linked to the deep feature model (ProcessOptions). Such feature model is generated from the textual concrete syntax shown in Listing 2.

6.2. Using language families

To use a deep model with PCs, like the one in Listing 1, it needs to be instantiated. Annotations in METADEPTH can attach actions to be triggered upon certain modelling events, like instantiation or value assignment. These actions are defined via a meta-object protocol (MOP) [CdL18, KR91]. This way, we have defined a MOP with actions for the PC annotations, to help instantiating deep model PLs. Specifically, when an element of a deep model with variability is instantiated (like ProcessModel in Listing 1), its PC is copied to the instance. Moreover, a constraint forbids instantiating a deep model PL if the associated deep feature model has features with potency 0.
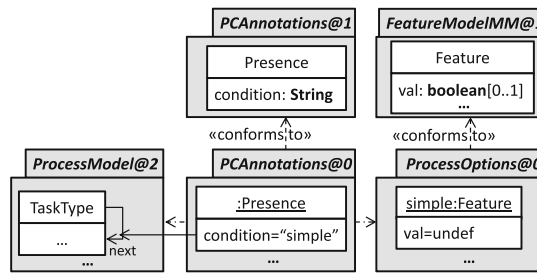
Fig. 22. Internal METADEPTH representation of a deep model PL.

Listing 3 displays a small instance of the deep model PL of Listing 1, as would be created by a modeller. Line 1 declares the model type (ProcessModel) and the model name (SoftwarePM). Then, the model declares two instances of TaskType called Requirements and Design. The former sets the value of field initial to *true*, while the latter sets it to *false* and declares a linguistic extension called description. Please note that, this model has potency 1, since the type has potency 2. This potency does not need to be explicitly specified, but it is calculated by the tool.

```
1  ProcessModel SoftwarePM {
2    TaskType Requirements{
3      initial = true;
4    }
5
6    TaskType Design {
7      initial = false;
8      description : String;
9    }
10 }
```

Listing 3: Using a deep model PL.

As mentioned, the MOP we have created for variability handling inserts the required PC annotations in the model as needed. Listing 4 shows the model with such annotations included (cf. lines 1, 4 and 8). This process is totally transparent to the modeller, who can recover such annotated model when it is displayed on METADEPTH's console by using command dump.

```
1  @Variability(model= "ProcessOptions")
2  ProcessModel SoftwarePM {
3    TaskType Requirements {
4      @Presence(condition= "initial")
5      initial=true;
6    }
7    TaskType Design {
8      @Presence(condition= "initial")
9      initial=false;
10     description:String;
11   }
12 }
```

Listing 4: Generated PC annotations.

In addition to instantiation, the user of the language family needs to configure the language. For this purpose, we have created a command called config to specialize a deep model PL via a configuration (see Listing 5). When the command is executed, the PCs attached to model elements are evaluated (partially if the configuration is partial), and they are removed if their value is *false*. The applied configuration (i.e., the boolean values assigned to the features) is stored in the deep feature model itself (cf. model ProcessOptions in Fig. 22).

```
1  config ProcessModel with { !initial, object }
```

Listing 5: Feature configuration.

The config command—when used without the with argument—can also be used to obtain the current configuration of a deep model PL. Overall, this simple example language already admits 16 total configurations, which can be succinctly represented as a PL, increasing its reuse possibilities.

Applying the configuration of Listing 5 eliminates the initial and next fields of TaskType in Listing 1, which are therefore eliminated from the instance models, like the one in Listing 4. Please note that by setting object to *true*, we are implicitly setting simple to *false*, since they are alternative features. Overall, the resulting deep model PL is shown in Listing 6.

```
1  @Variability(model="ProcessOptions")
2  Model ProcessModel@2 {
3    Node TaskType {
4      duration : int;
5      @Presence(condition="enactment")
6      rDuration : int;
7      @Presence(condition="actors")
8      perfBy : ActorKind;
9    }
10
11   @Presence(condition="actors")
12   Node ActorKind;
13
14   Node GatewayType {
15     src : TaskType[∗];
16     tar : TaskType[∗];
17   }
18 }
19 @Variability(model= "ProcessOptions")
20 ProcessModel SoftwarePM {
21   TaskType Requirements;
22   TaskType Design {
23     description:String;
24   }
25 }
```

Listing 6: Deep model PL after partial configuration.

6.3. Bottom-up extension of language families

As described in Sect. 6.3, another scenario of interest is the bottom-up creation of language families. In this scenario, extensions to both the feature model and the model are done at lower meta-levels, and then promoted to the top-level. For this purpose, we have created a command add feat, which adds features to the feature model. Listing 7 shows an example, which adds an optional feature named details, with potency 2, under the Tasks feature. The command updates the feature model, checking that the extension is possible according to Definition 14.

```
1  add opt feat details@2 under Tasks
```

Listing 7: Extending the feature model.

Then, the new features can be used within the PC formulae of the model elements. For example, we can use the introduced details optional feature to tag modelling elements that are useful to provide detailed insights of the different tasks of specific software process models, as Listing 8 depicts.

```
1  @Variability(model= "ProcessOptions")
2  ProcessModel SoftwarePM {
3    TaskType Requirements;
4    TaskType Design {
5      @Presence(condition="details")
6      description:String;
7    }
8    @Presence(condition="details")
9    Node Issue {
10     comments : String;
11     dsgnTasks : Design[∗];
12     reqTasks : Requirements[∗];
13   }
14 }
```

Listing 8: Extending a model PL at level 1.

In the listing, we have added the PC details to Design.description, and to clabject Issue. The latter is a linguistic extension that allows attaching comments to design or requirements tasks at the level below. At this point, the designer may realize that such an extension might be useful for other domains beyond software process modelling. Hence, we have created a command called promote to advance such extensions to instantiation, pulling them up to the upper meta-level. This process checks the pre-conditions in Theorem 5.4, required for the advancement to become possible. Listing 9 displays the resulting top-level model. In this model, field TaskType.description (line 10) was created, together with a new clabject IssueType (lines 17–20). The latter serves as a type for clabject Issue in Listing 8, and where field IssueType.tasks is the type for both Issue.dsgnTasks and Issue.reqTasks. Technically, the command promote uses two types of multi-level refactorings [dLG18]: createClabjectType and createFeatureType. The former creates a clabject type at the upper meta-level for a clabject linguistic extension, while the latter creates a field type at the upper meta-level for a field linguistic extension. Please note that we follow a naming convention for introduced clabject types (the name of the lower-level clabject followed by "Type"), while names of introduced reference types (like IssueType.tasks) need to be given by the modeller.

```
1   @Variability(model="ProcessOptions")
2   Model ProcessModel@2 {
3     Node TaskType {
4       duration : int;
5       @Presence(condition="enactment")
6       rDuration : int;
7       @Presence(condition="actors")
8       perfBy : ActorKind;
9       @Presence(condition="details")
10      description : String;
11    }
12
13    @Presence(condition="actors")
14    Node ActorKind;
15
16    Node GatewayType {
17      src : TaskType[*];
18      tar : TaskType[*];
19    }
20
21    @Presence(condition="details")
22    Node IssueType {
23      comments : String;
24      tasks : TaskType[*];
25    }
26  }
```

Listing 9: Extended deep model PL.

7. Related work

Next, we review related research coming from language product lines (Sect. 7.1), variability in multi-level modelling (Sect. 7.2) and software product lines (Sect. 7.3).

7.1. Language product lines

Some researchers have proposed increasing the reusability of modelling languages by incorporating SPL techniques (see [MGD+16] for a survey). For example, in [WHG+09], DSL meta-models can be configured using a feature model. In [PAA+16], the authors propose featured model types: meta-models whose elements have PCs, and with operations that are offered depending of the chosen variant. In [GdLCS20], meta-models can have variability, and their instantiability is analysed at the PL level. However, all these works only consider closed variability, while our work also supports open variability through instantiation (since we consider multi-level models).

In [BPRW20], the authors propose a framework—based on MontiCore [KRV10] and the principles of concern-oriented language development [CKM+18]—for defining language families with support for both open and closed variability. The framework relies on language components encapsulating syntax (through a grammar) and semantics (via code generators). Closed variability is achieved via a feature configuration that selects the components to be composed to form a language. For open variability, the composed language may contain extension points (e.g., expressing the need for an expression language) that other components need to satisfy, and parameters that can be assigned values. Other approaches follow similar ideas. For example, language definitions are modularized via roles in [WTZ09], while Melange [DCB+15] and Neverlang [VC15] also support modularization. In the first case, this is done via an algebra of operators for extending, restricting, and assembling separate language artefacts. In the second, by providing syntax definitions with placeholders, and modules that may implement language features.

Similar to the previous approaches, the closed variability in our approach is also achieved by configuring a feature model. However, instead of relying on required interfaces, extension points, or roles, we enable open variability via instantiation and linguistic extensions. We believe that both styles for open variability are complementary and suited for different scenarios. Our notion of open variability via refinement is better suited to specialize a generic language (e.g., a process modelling language) to specific domains (e.g., software process modelling, industrial process modelling). Instead, open variability via replacement of components is better suited to express alternative realizations for a concept (e.g., different types of expression languages).

The previous approaches [BPRW20, WTZ09, DCB+15, VC15] also consider the language semantics. METADEPTH is integrated with the Epsilon family of languages [PKR+09], which have been extended to work in a multi-level setting [dLGC15]. However, making these languages aware of variability via PLs is up to future work. Our plan in this aspect is to build on the ideas reported in [dLGCS18].

7.2. Multi-level modelling: variability and formalization

A plethora of multi-level modelling approaches and tools have emerged recently, like DeepTelos [JN16], FMMLx [Fra14], Melanee [AG16], MultEcore [MRS+18], MLT [FAGdC18], OMLM [IGSS18] and TOTEM [JdL20]. Some of them are based on deep characterization through potency [AG16, Fra14, IGSS18, MRS+18, JdL20], while others rely on powertypes [FAGdC18] or most-general instances [JN16]. None of them support variability based on feature models as we describe here. However, there have been some attempts to improve multi-level modelling with SPL techniques, which we describe next.

Reinhartz-Berger and collaborators [RSC15] present a preliminary proposal to support the configuration of classes with optional attributes. It is based on a *kernel language* with support for multiple meta-levels but lacking deep characterization. The proposal is incipient as it is neither formalized nor implemented. In [CFRS17], the authors analyse the limitations of feature models alone to describe a set of assets, and propose using multi-level models instead. As multi-level models have limitations to express variability—as described in Sec. 2.2—we propose to combine feature models and multi-level models.

Nesic and collaborators [NNG17] explore the use of MLT [FAGdC18] to reverse engineer sets of related legacy assets into PLs. MLT is a multi-level modelling approach based on powertypes and first order logic. In their work, the authors represent variability concepts like PCs and product groups within MLT models. This embedding may result in complex models where elements can represent either variability concepts or domain concepts. Instead, we separate PCs and feature models to avoid cluttering the multi-level model. Our goal is to

define highly reusable language families, for which we provide feature models to describe variability options, and offer the possibility to defer configurations; instead, the approach in [NNG17] lacks an explicit representation of feature models. Finally, we provide both a theory and a working implementation.

Other formalizations of potency-based multi-level modelling exist, like [RdLG⁺14] or the more recent [WMR20]. Those theories do not account for variability, but they could be extended with feature models, in a similar way as we do.

7.3. Software product lines

Our deferred configurations can be seen as a particular case of *staged configurations* [CHE05]. These permit selecting a member of the PL in stages, where each stage removes some choices. In our approach, the *potency* controls the level where the variability can be resolved. Staged configurations are also useful in software design reuse. In this setting, Kienzle and collaborators [KMCA16] propose Concern-Oriented Reuse, a paradigm where reusable modules (called *concerns*) define variability interfaces as feature models. The variability of a reused concern can be resolved partially, in which case, the undefined features are re-exposed in the interface of the resulting concern. We also support deferring the variability resolution, but composing deep model PLs is future work.

Taentzer and collaborators formalized model-based SPLs using category theory [TSSC17]. Different from ours, their formalization does not capture typing (it is within a single meta-level), while their morphisms can expand the feature model, but cannot be used to model partial configurations. Borba and collaborators have studied PL refinement, which adds new products maintaining the behaviour of existing ones [BTG12]. In our case, our variability extension morphisms preserve partial configurations.

To cope with large variability spaces, partitioning techniques can be applied to feature models to yield so-called multi-level feature models [CHE05, RPG⁺18]. However, the term multi-level does not refer to multiple levels of classification (as in our case), but to multiple partitions of a feature model.

In our work, we use F-morphisms (cf. Definition 8) to represent a partial configuration relation between two (deep) feature models, and EF-morphisms (cf. Definition 14) to represent an extension of a (deep) feature model. Related to this, syntactic and semantic differences between feature models have been studied in the PL community [AHC⁺12, DKMR19, TBK09]. In [TBK09] the authors present different types of relations between feature models, like refactoring (the products in both models remain the same), specialization (the set of products is reduced), and generalization (the set of products increase), along with algorithms based on SAT solving to compute them. In our case, F-morphisms correspond to specializations, while EF-morphisms are similar to generalizations (they in addition demand that invalid configurations cannot be extended to valid ones). Furthermore, our interest is in understanding whether extensions can be advanced to configurations (and to typing). In [AHC⁺12] the authors propose additional techniques for both syntactic and semantic differencing, to help in understanding and reasoning about differences. These differences can be combined with composition and decomposition operators. Open-world semantics for feature models, together with semantic diffs based on those semantics, are introduced in [DKMR19]. Such semantics includes all configurations—even containing features not belonging to the original feature model—which do not contradict the feature model formula. Hence, this notion is similar to our allowed extensions, and to the generalization relations in [TBK09].

Other modelling notations support variability. For example, Clafer [JSM⁺19] is an approach that unifies feature and class modelling. It supports both class and (partial) object models, feature models and their (partial) configurations and logic constraints. However, it does not support multi-level modelling or deep characterization. Similar to delta-oriented programming [SBB⁺10], in Δ-modelling, a core model (representing one product) is enriched with a set of changes (with application conditions) to capture further products [Sch10]. The approach has been proposed in combination with MDE, showing that model configuration and refinement (e.g., a component being refined by a set of classes) commute. This is in line with our Theorems 5.2 and 5.4, but we are interested in instantiation (instead of refinement), and need to incorporate potency for deep characterization. Therefore, in our case instantiation and specialization (configuration) do not commute, but the latter can be advanced to former. In addition, we have also studied the advancement of extension to both instantiation and specialization.

Within model-driven software product line engineering [CAK⁺05], some researchers have analysed techniques to manage variability across multiple models and artefacts [GW21, SPJ18]. In [GW21] the authors compare how different tools and approaches deal with the propagation of PCs across different models. They report that automated propagation is a feature that is scarcely supported. A multi-level model can be seen as a megamodel [BJRV05] made of a set of models related via instantiation relations. In our case, we do support automated propagation, for example, when instantiating a deep model PL (cf. Sect. 6.2), as well as when advancing extension to instantiation (cf. Sect. 6.3).

Table 1. Summary of approaches to variability in language and model-driven engineering (where *MM* stands for meta-model, *FM* for feature model, and *MLM* for multi-level model).

Approach	Variability support	Mechanism	Style	Meta-levels
White et al. [WHG⁺09]	Closed	MM + FM	Annotative	1 (meta-)
Featured model types [PAA⁺16]	Closed	MM + FM	Annotative	1 (meta-)
Meta-model PLs [GdLCS20]	Closed	MM + FM	Annotative	1 (meta-)
Monticore [BPRW20]	Open & closed	MM with extension points + FM	Compositional	1 (meta-)
Wendel et al. [WTZ09]	Open & closed	MM with Roles	Compositional	1 (meta-)
Neverlang [VC15]	Open & closed	Language Modules	Compositional	1 (meta-)
DeepTelos [JN16]	Open	MLM based on most general instances	Instantiation	arbitrary
FMMLx [Fra14]	Open	MLM based on instantiation levels	Instantiation	arbitrary
Melanee [AG16]	Open	MLM based on potency	Instantiation	arbitrary
MultEcore [MRS⁺18]	Open	MLM based on potency	Instantiation	arbitrary
MLT [FAGdC18]	Open	MLM based on powertypes	Instantiation + Classificat.	arbitrary
OMLM [IGSS18]	Open	MLM based on potency	Instantiation	arbitrary
TOTEM [JdL20]	Open	MLM based on potency	Instantiation	arbitrary
Our approach	Open & closed	MLM based on potency + FM	Instantiation + Annotative	arbitrary

In the programming world, Batory [Bat06, SB02] proposes mixin layers, a composition mechanism to add features to sets of base classes (so called two-level designs). Higher-level designs can be obtained by applying the same techniques. In [Bat06], these higher-level designs are called multi-level models. Again, the use of the term multi-level is different from ours, which refers to models related by classification relations.

As a summary, Table 1 classifies the approaches along their variability support (open, closed), the mechanisms involved (e.g., meta-models, feature models, etc.), the style (annotative, compositional, via instantiation or classification relations), and the meta-models on which the variability take place. The upper part of the table classifies approaches for language product lines, while the lower part contains approaches for multi-level modelling. Overall, our proposal is the first one adding variability to multi-level models with support for deep characterization via potency.

8. Conclusions and future work

In this paper, we have proposed a new notion of multi-level model PL to improve current reuse techniques for modelling languages. This is so as it permits both *open* variability (by successive instantiations leading to language refinements for specific domains), and *closed* variability (by selecting among a set of variants). We have presented a theory for the proper construction and use of language families. The theory contains results ensuring the proper interleave of instantiation, configuration and extension steps. The ideas have implemented on top of the multi-level modelling tool METADEPTH.

In the future, we plan to provide a categorical formalization of the theory, which would bring operations like intersection via common parts (pullbacks) and merging (pushouts) of deep model PLs. We would like to develop analysis techniques for multi-level model PLs, e.g., to check instantiability properties in the line of [GdLCS20]. Our goal is to make multi-level model PLs ready for MDE. This would entail the ability to define MDE services like transformations and code generators on multi-level model PLs. Technically, our plan is to use the Epsilon languages supported by METADEPTH, and follow ideas from existing works on PLs of transformations [dLGCS18], and transformation of PLs [SFR⁺14]. We would like to develop mechanisms for the assisted derivation of deep language families out of existing DSL meta-models, using as a basis the techniques for bottom-up modelling presented in Sect. 5.2. Finally, to proper model language families, we need to consider the concrete syntax as well. For this purpose, we plan to build on approaches to define graphical and textual syntaxes for multi-level models [Ger17, dLGC15], making them aware of closed variability through feature models (e.g., in the style of [GWG⁺20]).

Appendix

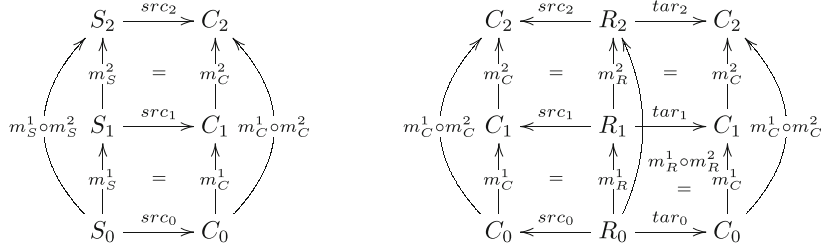In this appendix we provide the proof details of the lemmas and theorems proposed in the paper.

Fig. 23. Commutativity conditions for composition of D-morphisms.

Proof of Lemma 1 (D-morphism composition yields a D-morphism): In this lemma, we need to prove that, given two D-morphisms $m_1 = \langle d^1, m_C^1, m_S^1, m_R^1 \rangle \colon M_0 \to M_1$ and $m_2 = \langle d^2, m_C^2, m_S^2, m_R^2 \rangle \colon M_1 \to M_2$, their composition $m_2 \circ m_1 \colon M_0 \to M_2$ (with $m_2 \circ m_1 = \langle d^1 + d^2, m_C^2 \circ m_C^1, m_S^2 \circ m_S^1, m_R^2 \circ m_R^1 \rangle$) is a valid D-morphism. For this purpose, we proof the three conditions for D-morphisms of Definition 2, as follows:

1. We need to prove $p_0 + (d^1 + d^2) = p_2$. Since m_1 and m_2 are valid D-morphisms, we have $p_0 + d^1 = p_1$ and $p_1 + d^2 = p_2$, and so $p_0 + d^1 = p^2 - d_2$, and therefore $p_0 + (d^1 + d^2) = p_2$.

2. We need to prove $\forall\, e \in X_0 \bullet pot_0(e) + (d^1 + d^2) = pot_2(m_X^2 \circ m_X^1(e))$ (for $X \in \{C, S, R\}$). Since m_1 and m_2 are valid D-morphisms, we have $\forall\, e \in X_0 \bullet pot_0(e) + d^1 = pot_1(m_X^1(e))$ and $\forall\, e' \in X_1 \bullet pot_1(e') + d^2 = pot_2(m_X^2(e'))$ (for $X \in \{C, S, R\}$). Since the second property applies to all elements $e' \in X_1$, it also applies to $m_X^1(e)$, and so we have $\forall\, e \in X_0 \bullet pot_0(e) + d^1 = pot_1(m_X^1(e)) \wedge pot_1(m_X^1(e)) + d^2 = pot_2(m_X^2(m_X^1(e)))$. But then, $\forall\, e \in X_0 \bullet pot_0(e) + d^1 = pot_2(m_X^2 \circ m_X^1(e)) - d^2$ and so $\forall\, e \in X_0 \bullet pot_0(e) + (d^1 + d^2) = pot_2(m_X^2 \circ m_X^1(e))$ as required.

3. We need to proof that each function $m_C^2 \circ m_C^1$, $m_S^2 \circ m_S^1$, $m_R^2 \circ m_R^1$ commutes with functions src_i and tar_i. But this follows by the composition of commutative squares in set functions [EEPT06], as the outer squares of Fig. 23 shows.

Proof of Lemma 2 (F-morphism composition yields an F-morphism): In this lemma, we need to proof that, given two F-morphisms, $m_1 = \langle d^1, m_F^1, C^1 = \langle F_1^-, F_1^+ \rangle \rangle \colon DFM_0 \to DFM_1$ and $m_2 = \langle d^2, m_F^2, C^2 = \langle F_2^+, F_2^- \rangle \rangle \colon DFM_1 \to DFM_2$, their composition $m_2 \circ m_1 \colon DFM_0 \to DFM_2$ (defined as $\langle d^1 + d^2, \langle m_F^2(F_1^+) \cup F_2^+, m_F^2(F_1^-) \cup F_2^- \rangle, m_F^2 \circ m_F^1 \rangle$) is a valid F-morphism. For this purpose, we proof the three conditions for D-morphisms of Definition 8, as follows:

- We need to prove that $l_0 + (d^1 + d^2) = l_2$. Since both m_1 and m_2 are F-morphisms, we have that $l_0 + d^1 = l_1$ and $l_1 + d^2 = l_2$. Therefore $l_0 + d^1 = l_2 - d_2$ and so $l_0 + (d^1 + d^2) = l_2$ as required.

- We need to show that $m_F^2 \circ m_F^1$ is injective, and it is, since both m_F^2 and m_F^1 are injective and the composition of injective set functions is injective. In addition, we need to show that $\forall f \in F_0 \bullet pot_0(f) + d^1 + d^2 = pot_2(m_F^2 \circ m_F^1(f))$. Since m^1 and m^2 are F-morphisms, we have that $\forall f \in F_0 \bullet pot_0(f) + d^1 = pot_1(m_F^1(f))$ and $\forall f' \in F_1 \bullet pot_1(f') + d^2 = pot_2(m_F^2(f'))$. Since the last statement applies to all $f' \in F_1$, it also applies to $m_F^1(f)$, and so we have $\forall f \in F_0 \bullet pot_0(f) + d^1 = pot_1(m_F^1(f)) \wedge pot_1(m_F^1(f)) + d^2 = pot_2(m_F^2(m_F^1(f)))$ and therefore $\forall f \in F_0 \bullet pot_0(f) + d^1 + d^2 = pot_2(m_F^2 \circ m_F^1(f))$ as required.

- We need to proof that $\langle m_F^2(F_1^+) \cup F_2^+, m_F^2(F_1^-) \cup F_2^- \rangle \in CFG(FM_2)$, for which we need to check that $\Phi_2[m_F^2(F_1^+) \cup F_2^+/true, m_F^2(F_1^-) \cup F_2^-/false] \ncong false$. Because m_2 is F-morphism, $\Phi_2[F_2^+/true, F_2^-/false] \cong \Phi_1[F_1/m_F^2(F_1)] \ncong false$. Performing a further substitution for $\langle F_1^+, F_1^- \rangle$, if $\Phi_2[F_2^+/true, F_2^-/false][m_F^2(F_1^+)/true, m_F^2(F_1^-)/false] \cong false$, then $\Phi_1[F_1/m_F^2(F_1)][F_1^+/true, F_1^-/false] \cong false$. But since m_1 is F-morphism, this would mean that $\Phi_0[F_0/m_F^1(F_0)] \cong false$, which is not possible since FM_0 is a valid feature model according to Definition 5.

- We need to show that $m_F^2 \circ m_F^1(F_0) = F_2 \setminus (m_F^2(F_1^+) \cup F_2^+ \cup m_F^2(F_1^-) \cup F_2^-)$. Since both m_1 and m_2 are F-morphisms, we have that $m_F^1(F_0) = F_1 \setminus (F_1^+ \cup F_1^-)$ and $m_F^2(F_1) = F_2 \setminus (F_2^+ \cup F_2^-)$. From the first statement, we have that $F_1 = m_F^1(F_0) \cup F_1^+ \cup F_1^-$, and substituting in the second statement we have that $m_F^2(m_F^1(F_0) \cup F_1^+ \cup F_1^-) = F_2 \setminus (F_2^+ \cup F_2^-)$. Since $m_F^2(m_F^1(F_0) \cup F_1^+ \cup F_1^-) = m_F^2 \circ m_F^1(F_0) \cup m_F^2(F_1^+) \cup m_F^2(F_1^-)$

we have that $m_F^2 \circ m_F^1(F_0) = F_2 \setminus (F_2^+ \cup F_2^-) \setminus (m_F^2(F_1^+) \cup m_F^2(F_1^-)) = F_2 \setminus (F_2^+ \cup F_2^- \cup m_F^2(F_1^+) \cup m_F^2(F_1^-))$ as required.

- We need to show that $\Phi_2[F_2^+ \cup m_F^2(F_1^+)/true, F_2^- \cup m_F^2(F_1^-)/false] \cong \Phi_0[F_0/m_F^2 \circ m_F^1(F_0)]$. We have that $\Phi_1[F_1^+/true, F_1^-/false] \cong \Phi_0[F_0/m_F^1(F_0)]$ and $\Phi_2[F_2^+/true, F_2^-/false] \cong \Phi_1[F_1/m_F^2(F_1)]$. In the latter, statement, doing an additional substitution for $\langle F_1^+, F_1^- \rangle$ on both sides should preserve equivalence, and so we have $\Phi_2[F_2^+ \cup m_F^2(F_1^+)/true, F_2^- \cup m_F^2(F_1^-)/false] \cong \Phi_1[F_1/m_F^2(F_1)][m_F^2(F_1^+)/true, m_F^2(F_1^-)/false]$. But since $\Phi_1[F_1^+/true, F_1^-/false] \cong \Phi_0[F_0/m_F^1(F_0)]$, we have $\Phi_2[F_2^+ \cup m_F^2(F_1^+)/true, F_2^- \cup m_F^2(F_1^-)/false] \cong \Phi_0[F_0/m_F^2 \circ m_F^1(F_0)]$ as desired.

Proof of Lemma 3 (PL-morphism composition yields a PL-morphism): This lemma has two parts. In the first one, we need to prove that given PL-morphisms $m_1 = \langle m_1^D, m_2^F \rangle \colon DM_0 \to DM_1$ and $m_2 = \langle m_2^D, m_2^F \rangle \colon DM_1 \to DM_2$, their composition $m_2 \circ m_1 = \langle m_2^D \circ m_1^D, m_2^F \circ m_1^F \rangle \colon DM_0 \to DM_2$ is a valid PL-morphism.

From Definition 3 se have $m_2^D \circ m_1^D = \langle d^1 + d^2, m_C^2 \circ m_C^1, m_S^2 \circ m_S^1, m_R^2 \circ m_R^1 \rangle$, and according to Definition 9 we have $m_2^F \circ m_1^F = \langle d^1 + d^2, \langle m_F^2(F_1^-) \cup F_2^-, m_F^2(F_1^+) \cup F_2^+ \rangle, m_F^2 \circ m_F^1 \rangle$.

We need to prove:

$$\forall\, e \in C_0 \cup S_0 \cup R_0 \quad \bullet \quad \phi_2(m_2^D \circ m_1^D(e))[F_2^+ \cup m_F^2(F_1^+)/true, F_2^- \cup m_F^2(F_1^-)/false] \cong \phi_0(e)[F_0/m_F^2 \circ m_F^1(F_0)]$$

Since both m_1 and m_2 are PL-morphisms, we have that

(1) $\forall\, e \in C_0 \cup S_0 \cup R_0 \quad \bullet \quad \phi_1(m_1^D(e))[F_1^+/true, F_1^-/false] \cong \phi_0(e)[F_0/m_F^1(F_0)]$

and

(2) $\forall\, e' \in C_1 \cup S_1 \cup R_1 \quad \bullet \quad \phi_2(m_2^D(e))[F_2^+/true, F_2^-/false] \cong \phi_1(e')[F_1/m_F^2(F_1)]$

In the latter statement, since both terms are equivalent, performing an extra substitution $[m_F^2(F_1^+)/true, m_F^2(F_1^-)/false]$ also yields equivalence: $\forall\, e' \in C_1 \cup S_1 \cup R_1 \quad \bullet \quad \phi_2(m_2^D(e))[F_2^+ \cup m_F^2(F_1^+)/true, F_2^- \cup m_F^2(F_1^-)/false] \cong \phi_1(e')[F_1/m_F^2(F_1)][m_F^2(F_1^+)/true, m_F^2(F_1^-)/false]$. But because of (1), we have

$$\forall\, e \in C_0 \cup S_0 \cup R_0 \quad \bullet \quad \phi_2(m_2^D \circ m_1^D(e))[F_2^+ \cup m_F^2(F_1^+)/true, F_2^- \cup m_F^2(F_1^-)/false] \cong \phi_0(e)[F_0/m_F^2 \circ m_F^1(F_0)]$$

as desired.

For the second part of the lemma, we need to proof that, if m_1 and m_2 are specializations, so is $m_2 \circ m_1$. This means proving three conditions:

- Injectivity: Since both m_1^D and m_2^D are injective so is $m_2^D \circ m_1^D$.
- Level-preserving: Since both m_1 and m_2 are level-preserving, so is $m_2 \circ m_1$ since $d^1 = d^2 = 0 = d^1 + d^2$.
- Keeping elements with non-false PC: We require that $m_2 \circ m_1$'s co-domain contains exactly the elements $e \in C_2 \cup S_2 \cup R_2$ s.t. $\phi_2(e)[F_2^+ \cup m_F^2(F_1^+)/true, F_2^- \cup m_F^2(F_1^-)/false] \not\cong false$. The co-domain of the composition morphism $m_2 \circ m_1$ contains those elements, since m_2's co-domain includes each element e s.t. $\phi_2(e)[F_2^+/true, F_2^-/false] \not\cong false$, which is a larger set than the set of elements e making $\phi_2(e)[F_2^+ \cup m_F^2(F_1^+)/true, F_2^- \cup m_F^2(F_1^-)/false] \not\cong false$. In addition, the co-domain of $m_2 \circ m_1$ does not contain elements e s.t. $\phi_2(e)[F_2^+ \cup m_F^2(F_1^+)/true, F_2^- \cup m_F^2(F_1^-)/false] \cong false$, since m_1's co-domain does not contain any element e s.t. $\phi_1(e)[F_1^+/true, F_1^-/false] \cong false$.

Proof of Theorem 5.1 (Specialization morphisms for configurations): Given a deep model PL $DM = \langle M, DFM, \phi \rangle$ and a configuration $C = \langle F^+, F^- \rangle$ of DFM, we build $DM' = \langle M', DFM', \phi' \rangle$ as follows:

- M' has the same level as M, and contains each element e of M s.t. $\phi(e)[F^+/true, F^-/false] \not\cong false$. Functions src', tar' and pot' are restrictions of src, tar and pot to the elements in M'.
- $DFM' = \langle l, FM' = \langle F', \Phi' \rangle, pot' \rangle$, where $F' = F \setminus (F^+ \cup F^-)$, $\Phi' = \Phi[F^+/true, F^-/false]$, and pot' is the restriction of pot to F'.
- Function ϕ' is defined as follows: $\forall\, e \in C' \cup S' \cup R' \quad \bullet \quad \phi'(e) = \phi(e)[F^+/true, F^-/false]$.

Now we show that M' is a valid deep model according to Definition 1:

- To check that src' is well formed, we show that $\forall\, s \in S' \cup R'$, $src(s')$ is defined. By condition 1 in Definition 10, $\phi(s) \Rightarrow \phi(src'(s))$. This precludes the source of any $s \in S' \cup R'$ to be absent from C', since if $\phi(src'(s))[F^+/true, F^-/false] = false$, then $\phi'(s)[F^+/true, F^-/false] = false$.
- The well-formedness of tar' is shown like in the previous case.
- Function pot' satisfies conditions 1–3 of Definition 1, since pot satisfies them, and pot' is just a restriction of pot.

Now we show that DM' is a valid deep model PL according to Definition 10:

- M' and DFM' have the same level (l).
- The three conditions over ϕ' and pot' hold, since they hold for ϕ and pot.

Finally, we build a specialization PL-morphism $sp = \langle m^M, sp^F \rangle \colon DM' \to DM$ as follows:

- $m^M = \langle 0, inc_C^M, inc_S^M, inc_R^M \rangle$, where $X' \overset{inc_X^M}{\hookrightarrow} X$ (for $X = \{C, S, R\}$) are inclusion set morphisms,
- $sp^M = \langle 0, inc^F, C \rangle$, where $F' \overset{inc^F}{\hookrightarrow} F$ is an inclusion morphism.

For m_F, according to Definition 8, we need to show that: (i) $m_F(F') = F' = F \setminus (F^+ \cup F^-)$, which holds since F' was defined above as $F \setminus (F^+ \cup F^-)$; and (ii) $\Phi[F^+/true, F^-/false] \cong \Phi'[F'/inc_F(F')]$, which holds since Φ' was defined above as $\Phi[F^+/true, F^-/false]$.

Then, according to Definition 11, we need to show that $\forall\, e \in C' \cup S' \cup R' \bullet \phi(e)[F^+/true, F^-/false] \cong \phi'(e)$, which holds by construction. Finally, we need to show that $\forall\, e \in X \quad \bullet \quad (\phi(e)[F^+/true, F^-/false] \not\cong false \Leftrightarrow \exists\, e' \in X' \bullet sp_X^M(e') = e)$ (for $X = C, S, R$), which holds since we only include in DM' those elements for which $\phi(e)[F^+/true, F^-/false] \not\cong false$.

Now, we show that DM' is unique (up to isomorphism). To show that, assume we add a new clabject n to DM'. Let $sp_C^M(e)$ be the element in DM that e is mapped to. Because sp is specialization, we require $\phi(m_C^M(e))[F^+/true, F^-/\,false] \not\cong false$, and so $m_C^M(e)$ should have received a mapping from another node, since sp was a correct specialization. This means that we should map sp non-injectively, which is not possible by the definition of specialization. Adding new slots or references follows the same reasoning. Similarly, we could delete an element from DM', but in that case an element from DM with $\phi(m_C^M(e))[F^+/true, F^-/false] \not\cong false$ would not receive a mapping, which is not allowed by specialization morphisms. Finally, we cannot change the source or target of slots or references in DM', since then they would not commute properly, as required by correct D-morphisms. Please note that, while DM' is unique, there might be several (equivalent) ways to map it to DM. $\square$

Proof of Theorem 5.2 (Specialization can be advanced to instantiation): Let $C = \langle F^+, F^- \rangle$ be the configuration of the specialization PL-morphism $sp \colon DM_2 \to DM_1$. From DM_0 and C, we construct (uniquely) a deep model DM_3 and a specialization PL-morphism $sp' \colon DM_3 \to DM_0$ as described in the proof of Theorem 5.1. Then, we build a type PL-morphism $tp' = \langle tp'^D, tp'^F \rangle \colon DM_2 \to DM_3$ as follows:

- $tp'^D = \langle 1, tp_C^D \mid_{C_2}, tp_S^D \mid_{S_2}, tp_R^D \mid_{R_2} \rangle$, with $tp_X^D \mid_{X_2}$ the restriction of tp_X^D to set X_2 in DM_2 (for $X = \{C, S, R\}$).
- $tp'^F = \langle 1, tp_F^F \mid_{F_2}, C \rangle$ with $tp_F^F \mid_{F_2}$ the restriction of tp_F^F to set F_2.

D-morphism tp'^D is well defined because $\forall\, c \in C_2, \exists\, c' \in C_3$ s.t. $tp_C^D(sp_C^D(c)) = sp_C'^D(c')$. This is so as $\phi_1(sp_C^D(e))[F^+/true, F^-/false] \not\cong false$ due to Definition 11 of specialization PL-morphism. And now, since the configuration of tp is empty, we have $\phi_0(tp_C^D(sp_C^D(e))[F^+/true, F^-/false] \not\cong false$. This means that, according to Definition 11, this element is in the co-domain of $sp_C'^D$, and is assigned to c by $tp_C'^D$. The same reasoning applies to sets S_2 and F_2. Function $tp_F^F \mid_{F_2}$ is also well formed, since the same configuration C was used to derive DM_2 and DM_3. This reasoning also shows that $tp \circ sp = sp' \circ tp'$, as Theorem 5.2 demands.

Finally, please note that, once sp' is constructed, tp' is unique (while there can be several ways to build sp'). $\square$

Proof of Theorem 5.3 (Equivalent fully configured language): We need to check that given a chain of PL-morphisms:

$$DM_0 \overset{tp_1}{\Leftarrow} DM_1 \overset{sp_2}{\Leftarrow} DM_2 ... \overset{tp_n}{\Leftarrow} DM_n \overset{sp_{n+1}}{\Leftarrow} DM_{n+1}$$

where each tp_i is a type PL-morphism and each sp_i is an specialization PL-morphism, there is a unique deep model PL $DM_{FC} \in Der(DM_0)$ (called *fully configured model*) with a configuration made of all selected and unselected features of the configurations of $sp_2, ..., sp_{n+1}$, s.t. DM_{n+1} is an (indirect) instance of DM_{FC}.
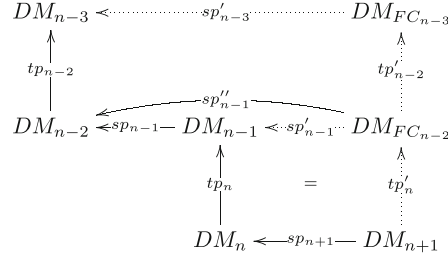
Fig. 24. Calculating the equivalent fully configured model.

We use the diagram in Fig. 24 for the proof. We start by applying Theorem 5.2 to advance sp_{n+1} to tp_n, obtaining the (unique) deep model PL $DM_{FC_{n-2}}$ and PL-morphisms $sp'_{n-1}\colon DM_{FC_{n-2}} \to DM_{n-1}$ (a specialization PL-morphism) and $tp'_n\colon DM_{n+1} \to DM_{FC_{n-2}}$ (a type morphism). Now, by Lemma 3 we can compose sp'_{n-1} and sp_{n-1}, which yields a specialization PL-morphism. Now, we apply again Theorem 5.2 to advance $sp_{n-1} \circ sp'_{n-1}$ to tp_n (cf. Fig. 24). This yields the (unique) deep model PL $DM_{FC_{n-3}}$ and PL-morphisms $sp'_{n-3}\colon DM_{FC_{n-3}} \to DM_{n-3}$ (a specialization PL-morphism) and $tp'_{n-2}\colon DM_{FC_{n-2}} \to DM_{FC_{n-3}}$ (a type morphism). We can iterate this procedure as needed. In the final result, we can obtain an indirect type morphism, by composing the calculated type PL-morphisms (tp'_n, tp'_{n-2},...). $\square$

Proof of Lemma 4 (EF-morphisms can be advanced to F-morphisms): We need to prove that given an F-morphism $m = \langle d, m_F, C\rangle\colon DFM_0 = \langle l_0, FM_0, pot_0\rangle \to DFM_1 = \langle l_1, FM_1, pot_1\rangle$ and an EF-morphism $DFM_0 \stackrel{m_e}{\Longrightarrow} DFM^{ext} = \langle l_0, FM^{ext}, pot^{ext}\rangle$, there is a deep feature model DFM_0^{ext} and morphisms $m' = \langle d, m'_F, C\rangle\colon DFM^{ext} \to DFM_0^{ext}$, $DFM_1 \stackrel{m'_e}{\Longrightarrow} DFM_0^{ext}$ s.t. $m'_e \circ m_F = m'_F \circ m_e$.

We construct $DFM_0^{ext} = \langle l_0^{ext}, FM_0^{ext} = \langle F_0^{ext}, \Phi_0^{ext}\rangle, pot_0^{ext}\rangle$ as follows:

- The level is set to $l_0^{ext} = l_0 + d$
- The feature set is set to $F_0^{ext} = F_0 \cup (F_1 \setminus m_F(F_0)) \cup (F^{ext} \setminus m_e(F_0))$, which is the disjoint union of F_1 and F^{ext} where the elements sharing a preimage in F_0 are identified, which is a pushout in the category of sets [EEPT06].
- The feature model formulae is set to $\Phi_0^{ext} = \Phi_1[m_F(F_0)/F_0] \vee \Phi^{ext}[m_e(F_0)/F_0]$ which is the disjunction of the formulas of DFM_1 and DFM^{ext} using the names of features in DFM_0.
- The potency is defined as:
  $\forall\, e \in F_0 \bullet pot_0^{ext}(e) = pot_1(m_F(e))$,
  $\forall\, e \in F_1 \setminus m_F(F_0) \bullet pot_0^{ext}(e) = pot_1(e)$, and
  $\forall\, e \in F^{ext} \setminus m_e(F_0) \bullet pot_0^{ext}(e) = pot^{ext}(e) + d$.

(The potency is taken from DFM_1 for those elements in F_0 or F_1, and from DFM^{ext} otherwise. Please note that the potency of mapped features in DFM_0 and DFM^{ext} is the same since m_e is an EF-morphism).

Now, we need to show that FM_0^{ext} is correct according to Definition 5, for which we need to show that Φ_0^{ext} is satisfiable. This holds since $\Phi_0^{ext} = \Phi_1 \vee \Phi^{ext}$ and both Φ_1 and Φ^{ext} are satisfiable. Then, we need to show that DFM_0^{ext} is correct according to Definition 7. This requires to show that the potency of each element is less than or equal to DFM_0's level. This is so for those elements receiving potency from pot^{ext}, since the level of DFM_0 and DFM^{ext} is the same, by Definition 15, and the level of DFM^{ext} is equal or larger ($l_0 + d$). The potency is less or equal than the level also for elements receiving the potency from pot^1, because the level of DFM_1 is $l_0 + d$, which is the level of DFM_0^{ext}.

Next, we need to show that there are commuting morphisms $m' = \langle d, m'_F, C\rangle\colon DFM^{ext} \to DFM_0^{ext}$, $DFM_1 \stackrel{m'_e}{\Longrightarrow} DFM_0^{ext}$ s.t. $m'_e \circ m_F = m'_F \circ m_e$.

First, we build F-morphism $m' = \langle d, m'_F, C\rangle\colon DFM^{ext} \to DFM_0^{ext}$ as follows:

- The depth d is equal to the depth d of F-morphism m.
- The mapping m'_F is defined as $\forall f \in F_0 \bullet m'_F(m_e(f)) = f$; $\forall f \in F^{ext} \setminus m_e(F_0) \bullet m'_F(f) = f$ (i.e., we build an identity mapping).

- The configuration C is equal to the configuration C of m.

  The F-morphism m' so constructed is valid according to Definition 8, since:

- The morphism depth fills the gap between the levels: $l^{ext} + d = l_0^{ext}$. This holds since $l^{ext} = l_0$ and $l_0^{ext} = l_0 + d$.
- Morphism m'_F is injective, since m_e is injective, and we built an identity for features in $F^{ext} \setminus m_F(F_0)$.
- The morphism depth d fills the gap between the potencies: $\forall f \in F^{ext} \bullet pot^{ext}(f) + d = pot_0^{ext}(m'_F(f))$. To show this, we split F^{ext} in $m_e(F_0)$ and $F^{ext} \setminus m_e(F_0)$ and check the property on these two sets:

  - On the one hand, $\forall f \in F_0 \bullet pot_0^{ext}(f) = pot_1(m_F(f))$ (by the way we have constructed DFM_0^{ext}), and $\forall f \in F_0 \bullet pot_0(f) + d = pot_1(m_F(f))$ (since m is an F-morphism). Therefore we have $\forall f \in F_0 \bullet pot_0^{ext}(f) = pot_0(f) + d$. Because m_e is an EF-extension, by Definition 14, we have that $\forall f \in F_0 \bullet pot_0(f) = pot^{ext}(m_e(f))$, and so $\forall f \in m_e(F_0) \bullet pot_0^{ext}(f) = pot^{ext}(m'_F(f)) + d$ as required.
  - On the other, by construction of DFM_0^{ext}, we have $\forall f \in F^{ext} \setminus m_e(F_0) \bullet pot_0^{ext}(f) = pot^{ext}(f) + d$, and because m'_F is an identity on those elements, $\forall f \in F^{ext} \setminus m_e(F_0) \bullet pot_0^{ext}(f) = pot^{ext}(m'_F(f)) + d$ as required.

- The configuration $C = \langle F^+, F^- \rangle$ of m'_F should be a configuration of DFM_0^{ext}. C is a configuration of DFM_0^{ext} if $\Phi_0^{ext}[F^+/true, F^-/false] \not\cong false$, Since C is a configuration of DFM_1, we have $\Phi_1[F^+/true, F^-/false] \not\cong false$, and so $\Phi_0^{ext}[F^+/true, F^-/false] \not\cong false$ as required.
- We need to show $m'_F(F^{ext}) = F_0^{ext} \setminus (F^+ \cup F^-)$. Since m is F-morphism, we have $m_F(F_0) = F_1 \setminus (F^+ \cup F^-)$, and so $F_1 \setminus m_F(F_0) = F^+ \cup F^-$. By construction, we have $F_0^{ext} = F_0 \cup (F_1 \setminus m_F(F_0)) \cup (F^{ext} \setminus m_e(F_0))$, and substituting we have $F_0^{ext} = F_0 \cup (F^+ \cup F^-) \cup (F^{ext} \setminus m_e(F_0))$. But by construction of m'_F we have that $F_0 \cup (F^{ext} \setminus m_e(F_0)) = m'_F(F^{ext})$, and substituting again $F_0^{ext} = m'_F(F^{ext}) \cup (F^+ \cup F^-)$. Solving, we find $F_0^{ext} \setminus (F^+ \cup F^-) = m'_F(F^{ext})$ as required.
- Finally, we need to show that $\Phi_0^{ext}[F^+/true, F^-/false] \cong \Phi^{ext}[F^{ext}/m'_F(F^{ext})]$. Using the remark in Definition 8, this is equivalent to require that any configuration $C' \in DFG(DFM^{ext})$ s.t. $C \leq C'$ is valid in DFM^{ext} iff it is valid in DFM_0^{ext}. By construction, $\Phi_0^{ext} = \Phi_1[m_F(F_0)/F_0] \vee \Phi^{ext}[m_e(F_0)/F_0]$. By the second term, any configuration $C' \in CFG(DFM^{ext})$ belongs to $CFG(DFM_0^{ext})$. Hence, any configuration $C' \in DFG(DFM^{ext})$ s.t. $C \leq C'$ of DFM^{ext} is valid in DFM_0^{ext}. Conversely, any non-valid configuration $C' \in DFG(DFM^{ext})$ s.t. $C \leq C'$ makes the second term false. However, the first term should also be false, since such configuration would make Φ_0 false (by the second part of condition 2 in Definition 14) and hence not equivalent to Φ_1 as required by m being an F-morphism.

Second, we build m'_e as follows: $f \in m_F(F_0) \Rightarrow m'_e(f) = m_F^{-1}(f)$, and $f \in F_1 \setminus m_F(F_0) \Rightarrow m'_e(f) = f$. Morphism m'_e so constructed is valid according to Definition 14, since (1) the potency pot_0^{ext} of all elements $e \in F_0 \cup F_1$ is taken from $pot^1(e)$, and (2) every configuration C of DFM_1 makes $\Phi_1[m_F(F_0)/F_0]$ true and hence a configuration of DFM_0^{ext}. In addition any invalid configuration C of DFM_1 makes $\Phi_1[m_F(F_0)/F_0]$ false, but also Φ_0 false (since m_F is an F-morphism, and $\Phi_1[F^+/true, F^-/false]$ and Φ_0 are equivalent). Hence, Φ^{ext} should also be false, since m_e is an EF-morphism, and therefore Φ_0^{ext} should also be false as required. $\square$

Proof of Theorem 5.4 (EPL-morphisms can be advanced to PL-morphisms): We need to prove that given a PL-morphism $m = \langle m^D, m^F \rangle \colon DM_0 \to DM_1$ and an EPL-morphism $m^e = \langle m_e^D, m_e \rangle \colon DM_0 \to DM^{ext}$, such that

1. $\forall e_i, e_j \in C_0 \cup S_0 \cup R_0 \bullet m^D(e_i) = m^D(e_j) \Rightarrow \phi^{ext}(m_e^D(e_i)) = \phi^{ext}(m_e^D(e_j))$
2. $\forall e \in C_0 \bullet \phi^{ext}(m_e^D(e)) = \phi_0(e)[F_0/m_e(F_0)]$.

then, we can build a deep model PL DM_0^{ext}, a PL-morphism $m' = \langle m'^D, m'^F \rangle \colon DM^{ext} \to DM_0^{ext}$ and an EPL-morphism $DM_1 \xrightarrow{m'^e} DM_0^{ext}$ with $m'^e = \langle m_e'^D, m'_e \rangle$ s.t. $m'_e \circ m^F = m'^F \circ m_e$ and $m'^D \circ m^D = m'^D \circ m_e^D$. First, we construct $DM_0^{ext} = \langle M_0^{ext}, DFM_0^{ext}, \phi_0^{ext} \rangle$ as follows:

- The model $M_0^{ext} = \langle p_0^{ext}, C_0^{ext}, S_0^{ext}, R_0^{ext}, src_0^{ext}, tar_0^{ext}, pot_0^{ext} \rangle$ is essentially a pushout in the category of graphs [EEPT06], constructed as follows:

  - The level $p_0^{ext} = p_0 + d$ is taken as the level of DM_0 plus the morphism's depth d.
  - $X_0^{ext} = X_1 \dot\cup X^{ext} |_{\equiv}$ (for $X = C, S, R$). Where $\dot\cup$ is the disjoint union and $\equiv$ is the smallest equivalence relation with $(m^D(e), m_e^D(e)) \in \equiv$ for all $e \in X_0$ (for $X = C, S, R$).

- $\forall\, e \in X_1 \bullet src_0^{ext}([e]) = [src_1(e)]$, and $\forall\, e \in X^{ext} \bullet src_0^{ext}([e]) = [src^{ext}(e)]$, for $X = S, R$ and where $[e]$ is the equivalence class of e in using relation $\equiv$ (i.e., the element in X_0^{ext} e is mapped to). Function tar_0^{ext} is constructed similarly.
  - The potency function is constructed as: $\forall\, e \in X_1 \bullet pot_0^{ext}([e]) = pot_1(e)$ (for $X = C, S, R$) and $\forall\, e \in X^{ext} \setminus m_e^D(X_0) \bullet pot_0^{ext}([e]) = pot^{ext}(e) + d$ (where d is the depth of morphism m^D).

- The deep feature model DFM_0^{ext} is constructed as in the proof of Lemma 4.
- The presence condition function is constructed as follows: $\forall\, e \in X_1 \setminus m^D(X_0) \bullet \phi_0^{ext}([e]) = \phi_1(e)$; $\forall\, e \in X^{ext} \setminus m_e^D(X_0) \bullet \phi_0^{ext}([e]) = \phi^{ext}(e)$; $\forall\, e \in X_0 \bullet \phi_0^{ext}([m^D(e)]) = \phi_1(m^D(e)) \wedge \phi^{ext}(m_e^D(e))$ (for $X = C, S, R$).

  Now, we need to show that DFM_0^{ext} is correct according to Definition 10, for which we need to check:

- DFM_0 and M_0 have the same level, which is the level of DM_0 plus the morphism's depth d.
- The function ϕ maps elements to a (non-false) propositional formula. This is so as neither ϕ_1 nor ϕ^{ext} map elements to false propositional formulae.
- The function ϕ satisfies the following three conditions:

  1. $\forall\, s \in S_0^{ext} \cup R_0^{ext} \bullet \phi_0^{ext}(s) \Rightarrow \phi_0^{ext}(src_0^{ext}(s))$. This holds since (a) it holds for both DFM_1 and DFM^{ext}, (b) the PC of elements in C_0 that are common in DFM_0 is not changed by m or m^e, according to condition 2 in this theorem, and (c) the PC of elements in $S^{ext} \cup R^{ext}$ can be strengthened (since m^e is an EPL-morphism). However, strengthening the premise of an implication preserves the implication (e.g., if we have $\phi^{ext}(s) \Rightarrow \phi_0(s)$ and $\phi_0(s) \Rightarrow \phi_0(src(s))$ then $\phi^{ext}(s) \Rightarrow \phi_0(src(s))$).
  2. $\forall\, r \in R_0^{ext} \bullet \phi_0^{ext}(r) \Rightarrow \phi_0^{ext}(tar_0^{ext}(r))$. This holds by the same reason as the previous property.
  3. $\forall\, e \in C_0^{ext} \cup S_0^{ext} \cup R_0^{ext},\ \forall\, v \in Var(\phi_0^{ext}(e)) \bullet pot_0^{ext}(v) \le pot_0^{ext}(e)$. For the elements mapped from DM_1, this holds as it holds for DM_1, and the potency is copied in those case. For those elements mapped from DM^{ext} and not common in DM_0, the potency of the elements in $C_0^{ext} \cup S_0^{ext} \cup R_0^{ext}$ is increased by d, just like the feature variables. Therefore, if it holds for DM^{ext}, it holds for DM_0^{ext}.

  Next, we construct the PL-morphism $m' = \langle m'^D, m'^F \rangle\colon DM^{ext} \to DM_0^{ext}$ as follows:

- The D-morphism $m'^D = \langle d', m'_C, m'_S, m'_R \rangle$ is built as follows:

  - The depth d' is taken as the depth of m^D (which is d).
  - Each element is mapped to its equivalent class under $\equiv$: $\forall\, e \in X^{ext} \bullet m'_X(e) = [e]$ (for $X = C, S, R$).

- m'^F is constructed as in the proof of Lemma 4.

  We need to show the three PL-morphism well-formedness conditions in Definition 2:

1. Property $p^{ext} + d' = p_0^{ext}$ holds since $p_0^{ext} = p_0 + d'$ and $p^{ext} = p_0$.
2. Property $\forall\, e \in X^{ext} \bullet pot^{ext}(e) + d' = pot_0^{ext}(m'_X(e))$ (for $X \in \{C, S, R\}$) holds by construction for those elements $e \in X^{ext} \setminus m_e^D(X_0)$. For those elements in $m_e^D(X_0)$, their potency is taken from pot_1. However, that potency is $pot_0 + d$, and since $pot^{ext} = pot_0$ and $d = d'$, we obtain the desired result.
3. Each function m'_C, m'_S, m'_R commutes with functions src_i and tar_i, which holds since DM_0^{ext} has been constructed as a pushout in the category of graphs [EEPT06].

  Then, we construct the EPL-morphism $m'^e = \langle m_e'^D, m'_e \rangle\colon DM_1 \to DM_0^{ext}$ as follows:

1. The D-morphism $m_e'^D = \langle d'^e, m_C'^e, m_S'^e, m_R'^e \rangle$ is built as follows:

   - The depth d'^e is 0.
   - Each element is mapped to its equivalent class under $\equiv$: $\forall\, d \in X_1 \bullet m_X'^e(d) = [d]$ (for $X = C, S, R$).

2. m'_e is constructed as in the proof of Lemma 4.

  Then, according to Definition 15 of EPL-morphism, we need to show that $\forall\, e \in C_1 \cup S_1 \cup R_1 \bullet \phi_0^{ext}(m'^e(e)) \Rightarrow \phi_1(e)[F_1/m'_e(F_1)]$. This holds for elements e in $X_1 \setminus m^D(X_0)$, since the PC is $\phi_0^{ext}([e]) = \phi_1(e)$, and therefore $\phi_1(e) \Rightarrow \phi_1(e)$. It also holds for elements e in $m^D(X_0)$, since their PC is $\phi_0^{ext}([m^D(e)]) = \phi_1(m^D(e)) \wedge \phi^{ext}(m_e^D(e))$, and so $\phi_1(m^D(e)) \wedge \phi^{ext}(m_e^D(e)) \Rightarrow \phi_1(m^D(e))$ as required.

Proof of Corollary 1 (Preservation of type and specialization PL-morphisms): First we proof that if m is a type PL-morphism, so is m'. PL-morphism $m = \langle m^D, m^F \rangle$ is type if m^D and m^F are types. Since the depth of m'^D and m'^F is that of m^D and m^F, then m' is also type.

Then, we assume m is specialization. Then, according to Definition 11, m^F is specialization, m^D is injective, level-preserving and satisfying $\forall e \in C_1 \cup S_1 \cup R_1 \bullet (\phi_1(e)[F_1^+/true, F_1^-/false] \not\cong false \Leftrightarrow \exists e' \in C_0 \cup S_0 \cup R_0 \bullet m^D(e') = e)$. If m^F is specialization, its depth is 0, and so is the depth of m'^F, and hence m'^F is type as well. If m^D is injective, so is m'^D because injectivity is preserved in pushouts in graphs (cf. fact 2.17 in [EEPT06]). If m^D is level preserving, so is m'^D since they have the same depth.

Finally, regarding the property on the PC, let's assume that for some element e in $X_0^{ext} \bullet \phi_0^{ext}(e)[F^+/true, F^-/false] \not\cong false$ (for $X = C, S, R$). If $\exists e' \in X_1$ with $m'^e(e') = e$ then since m is a specialization morphism, $\exists e'' \in X_0$ with $m^D(e'') = e'$. In such a case, by construction, $\exists e''' \in X^{ext}$ such that $m^e(e'') = e'''$, and $m^D(e''') = e$ as required. If instead $\nexists e' \in X_1$ with $m'^e(e') = e$, then by construction $\exists e' \in X^{ext}$ with $m'^D(e') = e$.

Conversely, let's assume that $\exists e' \in X^{ext}$ with $m'^D(e') = e$. Then we need to show that $\phi_0^{ext}(m'^e(e'))[F^+/true, F^-/false] \not\cong false$. By the condition in the corollary, $\phi^{ext}(e')[F^+/true, F^-/false] \not\cong false$. If $e' \in X^{ext} \setminus m^e(X_0)$ then $\phi_0^{ext}(e) = \phi^{ext}(e')$ and the condition holds. If $e' \in m^e(X_0)$ then $\phi_0^{ext}(e) = \phi_1(e'') \wedge \phi^{ext}(e')$. Since m is specialization, $\phi_1(e'')[F^+/true, F^-/false] \not\cong false$, and using the condition of the corollary, $\phi_0^{ext}(e)[F^+/true, F^-/false] \not\cong false$ as required.

Acknowledgements

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

[ACLF13] Acher M, Collet P, Lahire P, France RB (2013) FAMILIAR: a domain-specific language for large scale management of feature models. Sci Comput Program 78(6):657–681

[AG16] Atkinson C, Gerbig R (2016) Flexible deep modeling with melanee. In: Modellierung 2016, 2.-4. März 2016, Karlsruhe—Workshopband, pp 117–122

[AHC+12] Acher M, Heymans P, Collet P, Quinton C, Lahire P, Merle P (2012) Feature model differences. In: Advanced information systems engineering—24th international conference, CAiSE, volume 7328 of lecture notes in computer science, pp 629–645. Springer

[AK01] Atkinson C, Kühne T (2001) The essence of multilevel metamodeling. In: UML, volume 2185 of LNCS, pp 19–33. Springer

[AK02] Atkinson C, Kühne T (2002) Rearchitecting the UML infrastructure. ACM Trans Model Comput Simul 12(4):290–321

[AK08] Atkinson C, Kühne T (2008) Reducing accidental complexity in domain models. Softw Syst Model 7(3):345–359

[Atk97] Atkinson C (1997) Meta-modeling for distributed object environments. In: EDOC, pp 90–101. IEEE Computer Society

[Bat06] Batory DS (2006) Multilevel models in model-driven engineering, product lines, and metaprogramming. IBM Syst J 45(3):527–540

[BCW17] Brambilla M, Cabot J, Wimmer M (2017) Model-driven software engineering in practice. 2nd edn. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, San Rafael

[BJRV05] Bézivin J, Jouault F, Rosenthal P, Valduriez P (2005) Modeling in the large and modeling in the small. In: Model driven architecture, European MDA workshops: foundations and applications, MDAFA, volume 3599 of lecture notes in computer science, pp 33–46. Springer

[BPRW20] Butting A, Pfeiffer J, Rumpe B, Wortmann A (2020) A compositional framework for systematic modeling language reuse. In: MoDELS '20: ACM/IEEE 23rd international conference on model driven engineering languages and systems, pp 35–46. ACM

[BTG12]   Borba P, Teixeira L, Gheyi R (2012) A theory of software product line refinement. Theor Comput Sci 455:2–30

[CAK+05]   Czarnecki K, Antkiewicz M, Kim CHP, Lau S, Pietroszek K (2005) Model-driven software product lines. In: Companion to the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA, pp 126–127. ACM

[CdL16]   Córdoba-Sánchez I, de Lara J (2016) Ann: a domain-specific language for the effective design and validation of java annotations. Comp. Langs. Syst. Struct. 45:164–190

[CdL18]   Cuadrado JS, de Lara J (2018) Open meta-modelling frameworks via meta-object protocols. J Syst Softw 145:1–24

[CdLG12]   Cuadrado JS, de Lara J, Guerra E (2012) Bottom-up meta-modelling: an interactive approach. In: Model driven engineering languages and systems—15th international conference, MODELS, volume 7590 of lecture notes in computer science, pp 3–19. Springer

[CFRS17]   Clark T, Frank U, Reinhartz-Berger I, Sturm A (2017) A multi-level approach for supporting configurations: a new perspective on software product line engineering. In: ER Forum demo track, volume 1979 of CEUR workshop proceedings, pp 156–164. CEUR-WS.org

[CHE05]   Czarnecki K, Helsen S, Eisenecker UW (2005) Staged configuration through specialization and multilevel configuration of feature models. Softw Process Improv Pract 10(2):143–169

[CKM+18]   Combemale B, Kienzle J, Mussbacher G, Barais O, Bousse E, Cazzola W, Collet P, Degueule T, Heinrich R, Jézéquel J-M, Leduc M, Mayerhofer T, Mosser S, Schöttle M, Strittmatter M, Wortmann A (2018) Concern-oriented language development (COLD): fostering reuse in language engineering. Comput Lang Syst Struct 54:139–155

[DCB+15]   Degueule T, Combemale B, Blouin A, Barais O, Jézéquel J-M (2015) Melange: a meta-language for modular and reusable development of DSLs. In: SLE, pp 25–36. ACM

[DKMR19]   Drave I, Kautz O, Michael J, Rumpe B (2019) Semantic evolution analysis of feature models. In: Proceedings of the 23rd international systems and software product line conference, SPLC, pp 34:1–34:11. ACM

[dLG10]   de Lara J, Guerra E (2010) Deep meta-modelling with MetaDepth. In: TOOLS, volume 6141 of LNCS, pp 1–20. Springer

[dLG18]   de Lara J, Guerra E (2018) Refactoring multi-level models. ACM Trans Softw Eng Methodol 27(4):17:1–17:56

[dLG20]   de Lara J, Guerra E (2020) Multi-level model product lines—open and closed variability for modelling language families. In: Fundamental approaches to software engineering—23rd international conference, FASE, volume 12076 of lecture notes in computer science, pp 161–181. Springer

[dLGC15]   de Lara J, Guerra E, Cuadrado JS (2015) Model-driven engineering with domain-specific meta-modelling languages. Softw Syst Model 14(1):429–459

[dLGCS18]   de Lara J, Guerra E, Chechik M, Salay R (2018) Model transformation product lines. In: MoDELS, pp 67–77. ACM

[dLGS14]   de Lara J, Guerra E, Cuadrado JS (2014) When and how to use multilevel modelling. ACM Trans Softw Eng Methodol 24(2):12:1–12:46

[EEPT06]   Ehrig H, Ehrig K, Prange U, Taentzer G (2006) Fundamentals of algebraic graph transformation. Monographs in theoretical computer science. An EATCS Series. Springer

[FAGdC18]   Fonseca CM, Almeida JPA, Guizzardi G, de Carvalho VA (2018) Multi-level conceptual modeling: From a formal theory to a well-founded language. In: ER, volume 11157 of LNCS, pp 409–423. Springer

[Fra14]   Frank U (2014) Multilevel modeling—toward a new paradigm of conceptual modeling and information systems design. Bus Inf Syst Eng 6(6):319–337

[GdLCS20]   Guerra E, de Lara J, Chechik M, Salay R (2020) Property satisfiability analysis for product lines of modelling languages. IEEE Trans Softw Eng, to appear, 1–20

[Ger17]   Gerbig R (2017) Deep, seamless, multi-format, multi-notation definition and use of domain-specific languages. PhD thesis, University of Mannheim, Germany

[GPHS06]   González-Pérez C, Henderson-Sellers B (2006) A powertype-based metamodelling framework. Softw Syst Model 5(1):72–90

[GW21]   Greiner S, Westfechtel B (2021) On preserving variability consistency in multiple models. In: VaMoS'21: 15th international working conference on variability modelling of software-intensive systems, pp 7:1–7:10. ACM

[GWG+20]   Garmendia A, Wimmer M, Guerra E, Gómez-Martínez E, de Lara J (2020) Automated variability injection for graphical modelling languages. In: GPCE, pp 15–21. Association for Computing Machinery, New York, NY, USA

[IGSS18]   Igamberdiev M, Grossmann G, Selway M, Stumptner M (2018) An integrated multi-level modeling approach for industrial-scale data interoperability. Softw Syst Model 17(1):269–294

[JdL20]   Jácome-Guerrero SP, de Lara J (2020) *TOTEM*: reconciling multi-level modelling with standard two-level modelling. Comput Stand Interfaces 69:103390

[JN16]   Jeusfeld Manfred A, Neumayr B (2016) Deeptelos: multi-level modeling with most general instances. In: ER, volume 9974 of *LNCS*, pp 198–211

[JSM+19]   Juodisius P, Sarkar A, Mukkamala RR, Antkiewicz M, Czarnecki K, Wasowski A (2019) Clafer: lightweight modeling of structure, behaviour, and variability. Program J 3(1):2

[KC16]   Kühn T, Cazzola W (2016) Apples and oranges: comparing top-down and bottom-up language product lines. In: SPLC, pp 50–59. ACM

[KCH+90]   Kang K, Cohen S, Hess J, Novak W, Peterson A (1990) Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA

[KMCA16]   Kienzle J, Mussbacher G, Collet P, Alam O (2016) Delaying decisions in variable concern hierarchies. In: GPCE, pp 93–103. ACM

[KR91]   Kiczales G, Rivieres JD (1991) The art of the metaobject protocol. MIT Press, Cambridge

[KRV10]   Krahn H, Rumpe B, Völkel S (2010) Monticore: a framework for compositional development of domain specific languages. Int J Softw Tools Technol Transf 12(5):353–372

[KT08]   Kelly S, Tolvanen J-P (2008) Domain-specific modeling—enabling full code generation. Wiley

[Lan71]   Lane SM (1971) Categories for the working mathematician. Springer
[MGD⁺16]   Méndez-Acuña D, Galindo JA, Degueule T, Combemale B, Baudry B (2016) Leveraging software product lines engineering in the development of external dsls: a systematic literature review. Comput Lang Syst Struct 46:206–235
[MOF16]   MOF (2016) http://www.omg.org/spec/MOF
[MRB97]   Martin RC, Riehle D, Buschmann F (1997) Pattern Languages of Program Design 3. Addison-Wesley
[MRS⁺18]   Macías F, Rutle A, Stolz V, Rodríguez-Echeverría R, Wolter U (2018) An approach to flexible multilevel modelling. EMISA 13:10:1–10:35
[NC02]   Northrop L, Clements P (2002) Software product lines: practices and patterns. Addison-Wesley Longman Publishing Co., Inc., Boston
[NNG17]   Nesic D, Nyberg M, Gallina B (2017) Modeling product-line legacy assets using multi-level theory. In: SPLC, pp 89–96. ACM
[OMG04]   OMG (2004) UML human-usable textual notation. http://www.omgwiki.org/variability/doku.php
[PAA⁺16]   Perrouin G, Amrani M, Acher M, Combemale B, Legay A, Schobbens P-Y (2016) Featured model types: towards systematic reuse in modelling language engineering. In: MiSE@ICSE, pp 1–7. ACM, New York, NY, USA
[PBL05]   Pohl K, Böckle G, van der Linden FJ (2005) Software product line engineering: foundations, principles and techniques. Springer, Berlin
[PKR⁺09]   Paige RF, Kolovos DS, Rose LM, Drivalos N, Polack FAC (2009) The design of a conceptual framework and technical infrastructure for model management language engineering. In: ICECCS, pp 162–171. USA IEEE Computer Society, Washington, DC
[RdLG⁺14]   Rossini A, de Lara J, Guerra E, Rutle A, Wolter U (2014) A formalisation of deep metamodelling. Formal Asp Comput 26(6):1115–1152
[RPG⁺18]   Rabiser D, Prähofer H, Grünbacher P, Petruzelka M, Eder K, Angerer F, Kromoser M, Grimmer A (2018) Multi-purpose, multi-level feature modeling of large-scale industrial software systems. Softw Syst Model 17(3):913–938
[RSC15]   Reinhartz-Berger I, Sturm A, Clark T (2015) Exploring multi-level modeling relations using variability mechanisms. In: MULTI@MoDELS, volume 1505 of CEUR workshop proceedings, pp 23–32. http://ceur-ws.org
[SB02]   Smaragdakis Y, Batory DS (2002) Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans Softw Eng Methodol 11(2):215–255
[SBB⁺10]   Schaefer I, Bettini L, Bono V (2010) Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In: Proceedings of SPLC, volume 6287 of lecture notes in computer science, pp 77–91. Springer
[Sch06]   Schmidt DC (2006) Guest editor's introduction: model-driven engineering. Computer 39(2):25–31
[Sch10]   Schaefer I (2010) Variability modelling for model-driven development of software product lines. In: Proceedings of variability modelling of software-intensive systems (VaMoS), pp 85–92
[SFR⁺14]   Salay R, Famelis M, Rubin J, Sandro AD, Chechik M (2014) Lifting model transformations to product lines. In: ICSE, pp 117–128. ACM, New York, NY, USA
[SPJ18]   Strüber D, Peldszus S, Jürjens J (2018) Taming multi-variability of software product line transformations. In: Fundamental approaches to software engineering, 21st international conference, FASE, volume 10802 of lecture notes in computer science, pp 337–355. Springer
[TBK09]   Thüm T, Batory DS, Kästner C (2009) Reasoning about edits to feature models. In 31st international conference on software engineering, ICSE, pp 254–264. IEEE
[TSSC17]   Taentzer G, Salay R, Strüber D, Chechik M (2017) Transformations of software product lines: a generalizing framework based on category theory. In: MODELS, pp 101–111. IEEE Computer Society
[UML17]   UML 2.5.1 OMG specification (2017) http://www.omg.org/spec/UML/2.5.1/
[VBD⁺13]   Voelter M, Benz S, Dietrich C, Engelmann B, Helander M, Kats LCL, Visser E, Wachsmuth G (2013) DSL engineering—designing, implementing and using domain-specific languages. http://dslbook.org
[VC15]   Vacchi E, Cazzola W (2015) Neverlang: a framework for feature-oriented language development. Comput Lang Syst Struct 43:1–40
[WHG⁺09]   White J, Hill JH, Gray J, Tambe S, Gokhale AS, Schmidt DC (2009) Improving domain-specific language reuse with software product line techniques. IEEE Softw 26(4):47–53
[WMR20]   Wolter U, Macías F, Rutle A (2020) Multilevel typed graph transformations. In: Graph transformation—13th international conference, ICGT, volume 12150 of lecture notes in computer science, pp 163–182. Springer
[WTZ09]   Wende C, Thieme N, Zschaler S (2009) A role-based approach towards modular language engineering. In: Software language engineering, second international conference, SLE, volume 5969 of lecture notes in computer science, pp 254–273. Springer