E. Guerra, J. de Lara, M. Chechik and R. Salay, Property Satisfiability Analysis for Product Lines of Modelling Languages, in *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 397-416

**DOI:** https://doi.org/10.1109/TSE.2020.2989506

# Property Satisfiability Analysis for Product Lines of Modelling Languages

Esther Guerra, Juan de Lara, Marsha Chechik, and Rick Salay

**Abstract**—Software engineering uses models throughout most phases of the development process. Models are defined using modelling languages. To make these languages applicable to a wider set of scenarios and customizable to specific needs, researchers have proposed using product lines to specify modelling language variants. However, there is currently a lack of efficient techniques for ensuring correctness with respect to properties of the models accepted by a set of language variants. This may prevent detecting problematic combinations of language variants that produce undesired effects at the model level.
To attack this problem, we first present a classification of instantiability properties for language product lines. Then, we propose a novel approach to lifting the satisfiability checking of model properties of individual language variants, to the product line level. Finally, we report on an implementation of our proposal in the MERLIN tool, and demonstrate the efficiency gains of our lifted analysis method compared to an enumerative analysis of each individual language variant.

**Index Terms**—Model-Driven Engineering, Software Language Engineering, Product Lines, Meta-Modelling, OCL, Model Finding

## 1 INTRODUCTION

MODELLING is at the core of many software engineering activities. Models are built using the most suitable modelling language, either general-purpose (e.g., UML or Petri nets [1]) or domain-specific (DSL). In model-driven engineering (MDE) [2], the abstract syntax of a modelling language is described by a meta-model that defines domain primitives, their characteristics, relations and constraints. To express the latter, the OMG provides the standard Object Constraint Language (OCL) [3].

Modelling languages may admit variants to support multiple scenarios [4]. For example, we may offer variants of UML class diagrams for different purposes: while class operations or interfaces are not needed for analysis, we may include elements of particular programming languages, like generics or mixins, for detailed design. Building separate meta-models for every variant combination leads to a large meta-model set that, without proper support, is challenging to construct, analyse, manage and navigate. As an example of the real impact of this situation, Malavolta et al. [5] report more than 120 variations of architectural languages.

A way to simplify the management of all variants of a modelling language is to apply ideas from software product lines (SPLs) [6] to their construction, analysis and maintenance. SPLs permit expressing and managing collections of related software systems and their variability. They have been successfully applied to real-world problems within companies from varied domains, such as defence or the automotive industry [7], [8], [9], [10]. In language engineering, they have been used to handle variability [4], [11] of the language concrete syntax (i.e., visual representation), abstract syntax and semantics [11]. We focus on the abstract syntax (i.e., on meta-models) as the concrete syntax and semantics are defined on top of it. SPLs enable a compact representation of the language variants, providing an interface – *a feature model* [12] – for configuring a concrete language variant.

As an example, suppose we aim to build a meta-model product line (MMPL) to describe variants of Petri nets. Fig. 1 shows the meta-model of some of these variants. Variants (a) and (b) account for alternative realizations of tokens, either as attributes or objects; variant (c) describes hierarchical nets, where substitution transitions may contain places and transitions[1]; and variant (d) describes state-machine nets, where transitions have exactly one input and one output place. The targeted MMPL would encompass these meta-model variants and their consistent combinations, together with a feature model governing the presence/absence of meta-model elements depending on the selected features.

Since a meta-model determines the set of allowed models, guaranteeing its correctness is crucial. This includes ensuring that the set of instances of a meta-model satisfy desired properties like *instantiability*, i.e., having a non-empty instance set [13]. For this purpose, the most widely used approach is relying on constraint solving to assess whether the meta-model instances satisfy the expected properties. However, if we have an MMPL, the number of meta-models to analyse (and therefore the analysis time) can be exponential due to the combinatory nature of feature models. Hence, our aim is developing efficient techniques to ensure that some/all instance models of the meta-models within an MMPL have desired properties, such as meaningful combinations of objects, while ensuring absence of undesirable properties. Such techniques can be used by *language designers* to assess that a family of meta-models being developed is free of errors, and by *language users* to identify

- *E. Guerra and J. de Lara are with the Universidad Autónoma de Madrid, Spain. E-mail: {Esther.Guerra, Juan.deLara}@uam.es*
- *M. Chechik and R. Salay are with the University of Toronto, Canada. E-mail: {chechik, rsalay}@cs.toronto.edu*

*Manuscript received June 2019.*

1. While hierarchical nets may also consider fusion places, we leave this notion out of the example for simplicity of presentation.

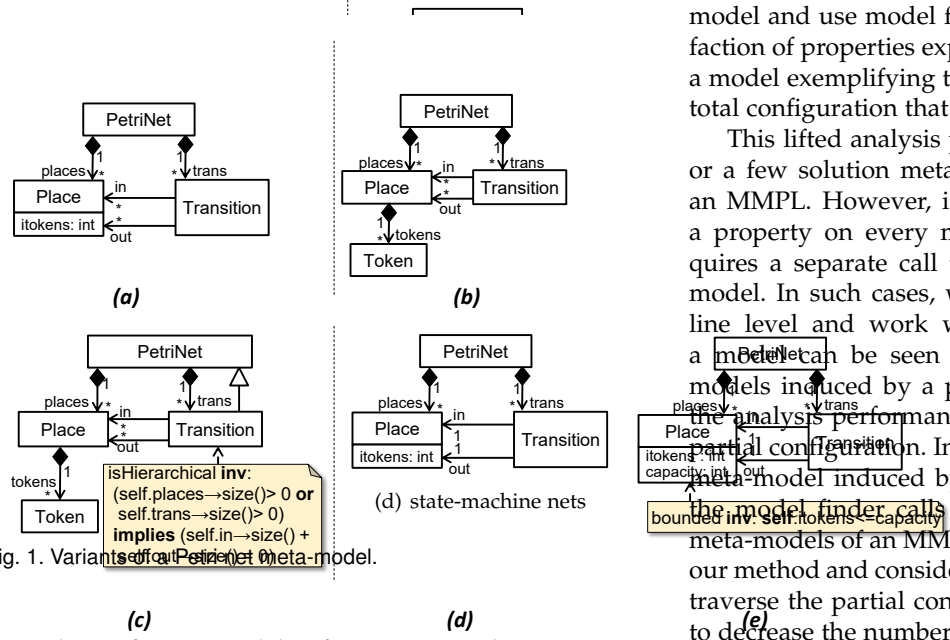



Fig. 1. Variants of a Petri net meta-model.

the subset of meta-models of an MMPL whose instances fulfil the properties needed for the user purposes. The latter is especially relevant to enable an informed meta-model selection on existing meta-model repositories such as [14]. Moreover, given that modelling and modelling languages are pervasive in software engineering, our techniques become generally applicable.

For example, we might like to ensure that no Petri net in any variant can have a negative number of tokens, which is *false* because some instances of meta-models (a) and (d) violate this property. However, assessing this property individually on each meta-model may be inefficient. Likewise, we would like to identify non-consistent combinations of features. Two variants conflict if their integrity constraints clash, preventing creating any model, or precluding the use of the primitives each variant offers. In the example, hierarchical and state-machine nets cannot be meaningfully combined, as nets with hierarchical transitions cannot be created: on the one hand, invariant isHierarchical restricts hierarchical transitions to have no input or output places, but on the other, state-machine nets require one input and one output place in all transitions.

Some approaches to MMPLs exist [4], [15], [16], [17], but we are not aware of techniques for ensuring a consistent combination of meta-model variants, or an effective analysis of properties pertaining to a subset of the languages of the family. While existing work on analysis of product lines of models [18] permits checking whether each product model conforms to its meta-model, our goal is to guarantee that the *meta-models* in a product line satisfy various *instantiability* properties. For this purpose, we cannot reuse existing analyses for model product lines as instantiability is an intrinsic characteristic of meta-models, not of models.

In this paper, we propose lifting the instantiability analysis from individual meta-models to product lines, to efficiently analyse satisfiability properties of the meta-models in an MMPL. To do so, we introduce a declarative notion of MMPL that is more amenable to automated analysis than the existing approaches [15], [16]. To check the satisfiability of properties, we embed the feature model within a meta-model and use model finders [19], [20] to analyse the satisfaction of properties expressed in OCL. The analysis returns a model exemplifying the property, its meta-model, and the total configuration that yields the meta-model.

This lifted analysis performs well when looking for one or a few solution meta-models among all meta-models in an MMPL. However, its efficiency is poor when checking a property on every meta-model of the MMPL, as it requires a separate call to the model finder for each meta-model. In such cases, we lift model typing to the product line level and work with partial configurations, so that a model can be seen as conformant to the set of meta-models induced by a partial configuration. This improves the analysis performance because the analysis can return a partial configuration. In this case, the result applies to every meta-model induced by the configuration, hence reducing the model finder calls needed to check a property for all meta-models of an MMPL. We provide correctness proofs of our method and consider two alternative search strategies to traverse the partial configurations, together with heuristics to decrease the number of model finder calls.

The paper also characterizes the space of instantiability properties for MMPLs, including both lifted properties and *mixed* properties that can refer to features of the feature model and be analysed for several meta-models of the MMPL. The approach is supported by the tool MERLIN, available at http://miso.es/tools/merlin. We use this tool to evaluate the performance gains of our lifted analysis based on total or partial configurations, in comparison to analysing each meta-model of the MMPL individually.

This paper extends our earlier work [21] as follows: (i) we propose a new theory based on partial configurations to improve the performance of the analysis, together with proofs of its correctness; (ii) we describe search strategies to traverse the partial configurations and heuristics to reduce the number of calls to a model finder; (iii) we provide tool support for the extended theory; (iv) we report on an evaluation based on eight MMPLs. The existing material from [21] is also expanded with proofs of all relevant theorems and additional examples.

*Paper organization*. Section 2 introduces MMPLs. Section 3 characterizes instantiability properties for MMPLs. Section 4 lifts satisfiability analysis to MMPLs. Section 5 shows how to optimize this analysis by using partial configurations. Section 6 presents tooling, and Section 7 evaluates its effectiveness. Finally, Section 8 discusses related work, and Section 9 concludes.

## 2 META-MODEL PRODUCT LINES

This section introduces our notion of MMPL, using the Petri nets example as an illustration. Our product lines follow an annotative approach combined with restricted types of transformations, which we call *modifiers* [21], [22]. Hence, in our approach, all meta-model variants are superimposed in a so-called 150% meta-model [23], [24], and their elements are annotated with conditions governing when they should appear in products. Modifiers express modifications on cardinalities and inheritance relations, triggered upon certain conditions. While some other approaches encode variability using richer, full-fledged programming languages [15], [16],

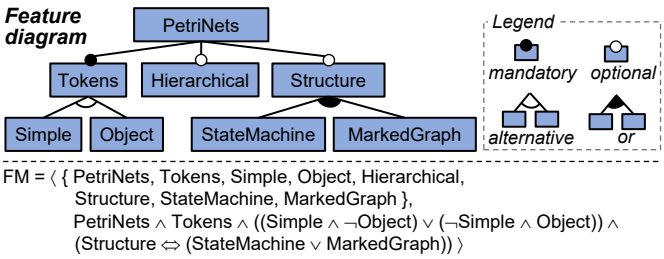



Fig. 2. Feature model for the Petri nets example.

[25], our work is applicable to any method that allows deriving a 150% meta-model.

First, we introduce the notion of *feature model* – a way to describe the variability space.

**Def. 1** (Feature model [21])**.** A *feature model* $FM = (F, \phi)$ consists of a set of features $F = \{f_1, ..., f_n\}$ and a propositional formula $\phi$ specifying the valid feature configurations.

**Example.** Fig. 2 shows a feature model describing the types of Petri nets in our running example (cf. Fig. 1). The upper part depicts the feature model using the feature diagram notation [12], while the bottom uses Def. 1. The feature model requires choosing a representation for tokens (Simple or Object, which are alternative), and optionally, any combination of Hierarchical, StateMachine and MarkedGraph can be selected. Marked graphs are nets where all places have exactly one input and one output transition. While we use this simple example for illustration, real SPLs are normally bigger and have a significant common part to foster reuse.

We base MMPLs on a standard notion of meta-model [26] with constraints.

**Def. 2** (Meta-model [21])**.** A *meta-model* is a tuple $MM = (C, FI, WC, I)$, where:

- $C$ is a set of *classes*, some of which may be *abstract*;
- $FI$ is a set of attributes and references called *fields*. Each field has a unique name, is owned by exactly one class, and has cardinality interval $[min..max]$, with $\infty$ used to denote unbound upper limits (i.e., $max = *$);
- $WC$ is a set of well-formedness constraints, called *invariants*, each of which is assigned to exactly one class; and
- $I \subseteq C \times C$ is the class inheritance relation by which a subclass has all fields and invariants of all its superclasses, and this relation must be acyclic.

Non-abstract classes define the types of objects that can appear in a model that is an instance of the meta-model. Fields of a class can be attributes of a primitive data type (e.g., int) or references pointing to a class. The cardinality of a field indicates the minimum and maximum number of values it can hold in an instance of the owning class. We define invariants using OCL [3]. These are declared in the context of a class, and evaluated on all of its instances. As in [21], we say that a meta-model is *well-formed* if its invariants are syntactically correct. Moreover, meta-models cannot have inheritance cycles. Intuitively, a model is said to *conform* to a meta-model if each object is typed by exactly one non-abstract class, each slot and link is correctly typed by compatible attributes and references, the cardinality of fields is preserved, and each object satisfies the invariants defined in its class and superclasses. We make this intuition more precise in Section 5.1.

*Remark.* Cardinalities can be seen as syntactic sugar, as they can be expressed by means of invariants; however, we prefer using cardinalities as they are easier to understand. For simplicity, Def. 2 omits other reference qualifications, like composition, which can also be expressed as invariants.

*Notation.* Given $f \in FI$, $target(f)$ refers to its target class if $f$ is a reference, or its data type if it is an attribute. We use the function $owner_{FI}\colon FI \to C$ to obtain the owner class of fields, and $owner^*_{FI} \subseteq FI \times C$ for the relation representing all classes that define or inherit a field. We use *ancs* for the reflexive-transitive closure of $I$.

**Example.** Fig. 1 shows four meta-models. The one in Fig. 1(c) has four non-abstract classes, five references with cardinality $[0..*]$, and one inheritance relation from class Transition to class PetriNet. Class Transition defines the invariant isHierarchical which demands every Transition with non-empty places or trans collections to have empty in and out collections.

Next, we define the notion of a *meta-model product line* to represent the meta-model variants of a language family.

**Def. 3** (Meta-model product line)**.** A *meta-model product line* is a tuple $MMPL = (FM, MM, \Phi, M_C, M_I)$, where[2]:

- $FM = (F, \phi_{MMPL})$ is a feature model;
- $MM = (C, FI, WC, I)$ is a meta-model, called the 150% meta-model (*150MM* in short);
- $\Phi = (\Phi_C, \Phi_{FI}, \Phi_{WC})$ is a tuple of mappings from the feature model to the *150MM*. Each mapping $\Phi_X$, for $X \in \{C, FI, WC\}$, consists of pairs $\langle x, \Phi_x \rangle$ mapping an element (a class, a field or an invariant) $x \in X$ to a propositional formula $\Phi_x$ over features, called its *presence condition* (PC). The PC of a field $f$ must be stronger than that of its owning class $C_i$ (i.e., an implication $\Phi_f \Rightarrow \Phi_{C_i}$ is required), and same for invariants;
- $M_C = (\mu_{min}, \mu_{max})$ is a tuple of sets of *cardinality modifiers*. The set $\mu_{min}$ (resp. $\mu_{max}$) consists of tuples $\langle f, m, \Phi_{min} \rangle$ mapping a field $f \in FI$ to a new minimum (resp. maximum) cardinality $m$ whenever the first-order formula $\Phi_{min}$ (resp. $\Phi_{max}$) is *true*;
- $M_I = (\mu_{add}, \mu_{del})$ is a tuple of sets of *inheritance modifiers*. The set $\mu_{add}$ (resp. $\mu_{del}$) consists of tuples $\langle C_{sub}, C_{super}, \Phi_{add} \rangle$ adding (resp. deleting) an inheritance relation $(C_{sub}, C_{super})$ when the first-order formula $\Phi_{add}$ (resp. $\Phi_{del}$) is *true*.

The *150MM* collects all elements appearing in the meta-models of the MMPL. Its elements are decorated with PCs which are boolean formulas that may use the features $F$ of *FM*. For deriving a concrete product, the elements whose PC is *false* are removed from the *150MM*. In other words, an element (class, field, invariant) becomes present in a meta-model when its PC evaluates to *true*.

**Example.** Fig. 3 shows the *150MM* for the running example. Conceptually, it is made of all superimposed meta-model variants of interest (in our example, those in Fig. 1 and some others). It is decorated with PCs (blue boxes on top

2. We use MMPL as the acronym of meta-model product line, and *MMPL* (in italics) to denote the tuple name.

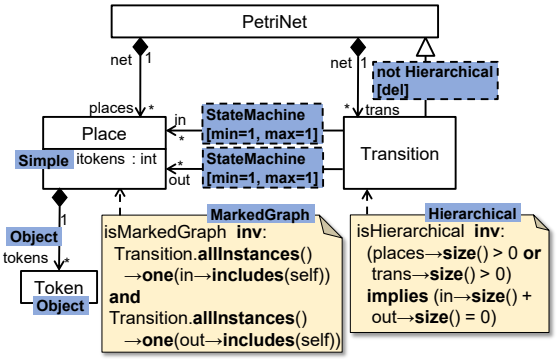



Fig. 3. *150MM* annotated with PCs and modifiers.

⇒ SEMANTIC ERROR a constraint requiring positive tokens is missing for Simple
⇒ SEMANTIC ERROR (after adding presence Condition Hierarchical to invariant isHierarchical): there is no model
illustrating Hierarchical and StateMachine relevance to inheritance and references: transitions
(to illustrate the inheritance), in which case in/out should be empty; however, StateMachine requires transitions to have
1 input and 1 output place (matches in input/output)

of classes, fields and invariants) and modifiers (dashed blue boxes) on top of inheritance relations and references).

For example, the attribute Place.itokens appears when its PC Simple is *true*, while the inheritance relation from Transition to PetriNet is deleted (i.e., it does not appear) when ¬Hierarchical is satisfied. References Transition.in and Transition.out have two cardinality modifiers each, making the *min* and *max* cardinality of the references equal to 1 when StateMachine is *true*.

This *150MM* is well-formed, i.e., all invariants are syntactically correct. When StateMachine is *true*, references Transition.in and Transition.out become mono-valued (i.e., their *max* cardinality becomes 1); still, the invariant isHierarchical remains valid because in→**size**() is a shorthand for in→**asSet**()→**size**(), which converts in into a set and permits applying operator **size**() to the result. Had we omitted the inheritance relation from Transition to PetriNet and used a modifier to add this relation when Hierarchical is *true*, the invariant isHierarchical would have been syntactically incorrect as the class Transition would lack fields trans and places.

*Remark.* Def. 3 uses modifiers to change element characteristics. Modifiers can be seen as a very restricted form of transformational PL approach [15], which we combine with the mechanisms of negative variability. We currently support modifiers for field cardinalities and inheritance, and leave the definition of other modifiers (e.g., for the abstractness of classes or the type of fields) as future work. Anyhow, modifiers are not strictly necessary as they can be emulated with PCs. For example, given a modifier of the cardinality of a field when $\Phi_{min}$ is *true*, it can be represented as an invariant demanding the satisfaction of the modifier cardinality and PC $\Phi_{min}$. However, we use modifiers as this allows reusing existing meta-model editors to build *150MMs*. These editors expect meta-models with correct structure, but are more permissive with the OCL invariants, typically handled as uninterpreted strings. Without modifiers, we would need editors able to accept inheritance cycles. Instead, our approach allows attaching any number of modifiers to classes and fields, e.g., as annotations, which are supported by widely used meta-model editors like the Ecore tree editor [27], OCLinEcore [28], Xcore [29] or EMFatic [30]. As modifiers can change the inheritance relations, the *150MM* may include invariants referring to fields that become inherited only when a modifier is triggered. Hence, *150MMs* are meta-models as per Def. 2, but they do not need to be well-formed.

A feature model defines a set of valid *total* configurations: assignments of truth values to features that satisfy the feature model. We also consider *partial* configurations with the standard meaning, i.e., where the value of some features is undefined. We will use these latter configurations in Section 5. Total configurations permit selecting one meta-model from the product line, containing all elements in the *150MM* whose PC is *true*.

**Def. 4** (Configuration)**.** Given a feature model $FM = (F, \phi_{MMPL})$, a feature configuration $\rho = \langle \rho^+, \rho^- \rangle$ is a tuple of two disjoint feature sets, with $\rho^+$ the features whose value is set to *true*, and $\rho^-$ the features whose value is *false*, s.t.:

1) $\rho^+ \cup \rho^- \subseteq F$;
2) $\phi_{MMPL}[true/\rho^+, false/\rho^-] \not\cong false$

We write $\rho^u = F \setminus (\rho^+ \cup \rho^-)$ for the set of undefined features. We call $\rho$ *total* if $\rho^u = \emptyset$, and call it *partial* otherwise. We use $P(FM)$ for the set of all total configurations of *FM*, and $\tilde{P}(FM)$ for the set of all its partial configurations.

*Remark.* In Def. 4, item 2 checks that the feature model formula $\phi_{MMPL}$ is not equivalent to *false* when we substitute all features in $\rho^+$ by *true*, and those in $\rho^-$ by *false*. In the following, we use a tilde ($\tilde{\rho}$) to mark partial configurations.

**Example.** The feature model in Fig. 2 admits 16 valid total configurations, including $\rho_0$ = $\langle$ {PetriNets, Tokens, Simple}, {Object, Hierarchical, Structure, StateMachine, MarkedGraph} $\rangle$, $\rho_1$ = $\langle$ {PetriNets, Tokens, Object}, {Simple, Hierarchical, Structure, StateMachine, MarkedGraph} $\rangle$, $\rho_2$ = $\langle$ {PetriNets, Tokens, Object, Hierarchical}, {Simple, Structure, StateMachine, MarkedGraph} $\rangle$, and $\rho_3$ = $\langle$ {PetriNets, Tokens, Simple, Structure, StateMachine}, {Object, Hierarchical, MarkedGraph} $\rangle$.

In addition, it admits 110 partial configurations. For example, $\tilde{\rho}_4 = \langle \emptyset, \emptyset \rangle$ leaves all features undefined, while $\tilde{\rho}_5$ = $\langle$ {PetriNets, Structure, StateMachine}, {Hierarchical, MarkedGraph} $\rangle$ sets features PetriNets, Structure and StateMachine to *true*; Hierarchical and MarkedGraph to *false*; and the rest to *undefined*.

Algorithm 1 describes how to derive a meta-model $MM_\rho$ from an MMPL *MMPL*, given a total configuration $\rho$. If an element has no PC, the algorithm assumes that its PC is *true*. Lines 1–2 select the *150MM* elements whose PC evaluates to *true*. Lines 3–5 add the inheritance relations that have no modifiers and are defined between classes selected by the configuration. Lines 6–9 evaluate the *min* and *max* cardinality modifiers, updating the cardinality of fields if needed. The remaining lines handle inheritance modifiers. An inheritance relation is added to $MM_\rho$ when it has no modifier (as seen in lines 3–5), its *add* modifier evaluates to *true* (lines 10–12), or its *del* modifier evaluates to *false* (lines 13–15).

*Notation.* We write *Prod(MMPL)* for the set of all derivable meta-models from an *MMPL*.

**Example.** Fig. 1 shows four meta-models derived from the MMPL in Figs. 2 and 3 using configurations $\rho_0$ (Fig. 1(a)), $\rho_1$ (Fig. 1(b)), $\rho_2$ (Fig. 1(c)), and $\rho_3$ (Fig. 1(d)).

Def. 3 does not require the *150MM* to be well-formed. However, a reasonable requirement for an MMPL is that every derived meta-model is well-formed. We call these MMPLs *well-formed*.

**input :** $MMPL$, $\rho \in P(FM)$
**output:** A meta-model $MM_\rho$
1 **forall** $e \in MM_C \cup MM_{FI} \cup MM_{WC}$ **do**
2  **if** $\Phi_e[true/\rho^+, false/\rho^-]$ **then add** $e$ **to** $MM_\rho$
3 **forall** $(C_{sub}, C_{super}) \in MM_I$ with no modifier **do**
4  **if** $C_{sub}$ and $C_{super}$ are included in $MM_\rho$ **then**
5  **add** $(C_{sub}, C_{super})$ **to** relation $I$ in $MM_\rho$
6 **forall** $\langle f, m, \Phi_{min}\rangle \in \mu_{min}$ **do**
7  **if** $\Phi_{min}[true/\rho^+, false/\rho^-]$ **then set** $f.min = m$
8 **forall** $\langle f, m, \Phi_{max}\rangle \in \mu_{max}$ **do**
9  **if** $\Phi_{max}[true/\rho^+, false/\rho^-]$ **then set** $f.max = m$
10 **forall** $\langle C_{sub}, C_{super}, \Phi_{add}\rangle \in \mu_{add}$ **do**
11  **if** $\Phi_{add}[true/\rho^+, false/\rho^-]$ **then**
12  **add** $(C_{sub}, C_{super})$ **to** relation $I$ in $MM_\rho$
13 **forall** $\langle C_{sub}, C_{super}, \Phi_{del}\rangle \in \mu_{del}$ **do**
14  **if** $\neg\Phi_{del}[true/\rho^+, false/\rho^-]$ **then**
15  **add** $(C_{sub}, C_{super})$ **to** relation $I$ in $MM_\rho$

**Algorithm 1:** Meta-model derivation. Not from Defs. 2, 3 and 4.

**Def. 5** (Well-formed MMPL)**.** A product line $MMPL$ is *well-formed* iff every $MM \in Prod(MMPL)$ is well-formed.

In [21], we showed an efficient way to assess whether a given MMPL is well-formed by lifting the syntactic analysis to the MMPL level. In this paper, we assume well-formed MMPLs and refer to [21] for details.

## 3 Instantiability Properties of MMPLs

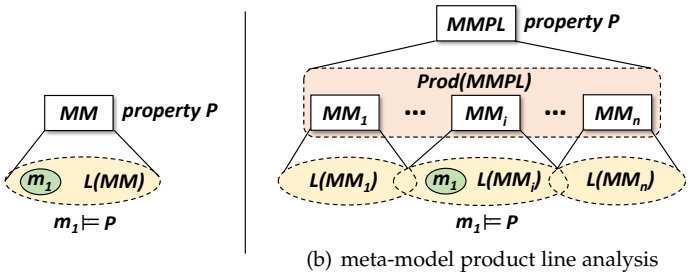
This section motivates the need for instantiability analysis at the level of PLs (Section 3.1) and characterizes instantiability properties for MMPLs (Section 3.2).

### 3.1 Instantiability Analysis for MMPLs

When designing a language, a syntactically correct meta-model is not sufficient, but in addition, the set of accepted models must be the one that the designer intends. Hence, to validate a meta-model $MM$, we need to check that its language $L(MM)$ does not contain models considered invalid, and does not miss models considered valid.

Fig. 4(a) shows how this validation process is automated using *model finders* [13], [19], [20], [31], [32]. These are tools that receive a meta-model (including its invariants) and a set of model properties as input, and use constraint solving to find a model that conforms to the meta-model and exhibits the specified properties. This search is typically bound to models up to a given size. Model finders like USE Validator [20] and EMFtoCSP [31] allow expressing the desired model properties in OCL.

Hence, to check a property $P$ on the instances of a meta-model, the engineer expresses the property using OCL and uses a model finder to look for a model satisfying $P$ within the search bounds. If a model is found, then it is a witness of the property satisfiability by some meta-model instance; otherwise, the analysis is inconclusive (i.e., it shows that no model satisfies $P$ within the search bounds, but there may be some out of these bounds). Checking whether all models in $L(MM)$ satisfy $P$ is done by checking that no model satisfies $\neg P$. In the simplest case, $P$ may be empty, and the analysis then confirms whether the meta-model is instantiable (i.e., $L(MM)$ is not empty) and therefore has no conflicting invariants.




(a) (b) (b) meta-model product line analysis

Fig. 4. Analysis of meta-model instance properties.

The analysis of instance properties is a useful and accepted meta-model validation technique [13], [20], [31], [32], [33]. While it is applicable to each meta-model in an MMPL, performing model finding on each meta-model may be time-consuming. Therefore, we propose lifting instantiability analysis to the PL level, as shown in Fig. 4(b). This involves being able to express properties independently of the specific meta-model (hence promoting conciseness) and their efficient checking without generating and checking each MMPL meta-model. For this purpose, the rest of the section characterizes analysis of meta-model instance properties for MMPLs, and Section 4 focuses on their efficient analysis.

### 3.2 Classifying Instantiability Analyses for MMPLs

Fig. 5 uses a feature model to characterize the options when analysing instance properties for some/all meta-models of an MMPL. The feature model is structured along three orthogonal aspects. The first one concerns the specification of the *property* $P$ to analyse. The second one targets the analysis *scope*, i.e., the set of meta-models or models subject to the analysis. The last aspect is the format of the analysis *result*, which may be either a set of witness artefacts or a simple yes/no assessment. Next, we describe the space of options for each aspect.

**1) Mixed property.** The property $P$ to analyse may refer only to structural meta-model elements, or to features of the feature model as well. We call the latter properties *mixed*. For example, the following mixed property checks whether any model that only contains transitions with one input place belongs to configurations where feature StateMachine is *true*: Transition.**allInstances**()→ **forAll**(t | t.in→**size**() = 1) **implies** StateMachine.

**2) Property satisfiability.** Given a property $P$, we may want to analyse whether the MMPL has some meta-model of which at least one model satisfies $P$, or for which no model satisfies $P$, or where all its models satisfy $P$. These three options correspond to features $\text{exists}_m$, $\text{notExists}_m$ and $\text{forAll}_m$ in Fig. 5.

**3) Configuration scope.** A property $P$ may be analysed just within a certain *configuration scope*, e.g., if the property only applies to meta-models of some configurations in the MMPL. This scope may be explicitly defined, or be inferred from the property. For example, given the property Place.**allInstances**()→**forAll**(p | p.itokens = 0), we can infer that it applies just to configurations that select the feature Simple, as the attribute itokens is only present when this feature is *true*.

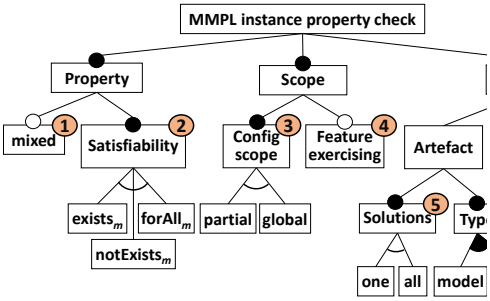
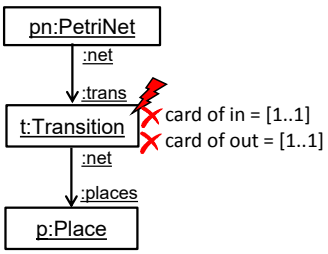
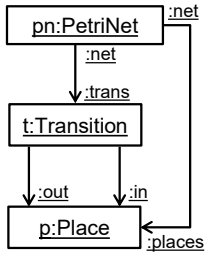



Fig. 5. Options for MMPL instance property analysis




(b) State-machine transition

Fig. 6. Vacuous feature combinations. (a) Hierarchical transitions make state-machine invariants fail. (b) State-machine transitions do not exercise the elements introduced by feature Hierarchical.

**4) Feature exercising.** Sometimes, it is us
models that contain instances of the eleme
selected features to illustrate the meta-mo
a configuration makes available, to compa
or to reason about feature interactions. Fo
this option, models of configurations th
Object need to contain at least one instance
this class is only added when this feature

**5) Solutions.** Given an MMPL, the analysi
meta-models having instances that satisfy
of those meta-models.

**6) Result type.** The analysis may return a model $m$ satisfying $P$, the feature configuration that produces the meta-model of which $m$ is an instance, or both.

**7) Assessment.** The aim of the analysis may be assessing whether some, all or no meta-model of the MMPL satisfies $P$. These three analysis questions correspond to features exists$_{MM}$, forAll$_{MM}$ and notExists$_{MM}$ in Fig. 5. The answer to these questions can be yes or no.

This space of options allows expressing many types of instantiability analyses for the meta-models of an MMPL. Table 1 illustrates some of them, though other analyses are possible provided that they follow the rules captured by the feature model in Fig. 5. In each case, the table lists the selected analysis options, a description of the analysis, and an example.

Analyses 1–3 address the instantiability (i.e., the existence of models) of the meta-models in an MMPL, and leave the property $P$ empty. The first analysis seeks the configuration of one instantiable meta-model; the second, an example model of one instantiable meta-model; and the third, an example model of each instantiable meta-model.

In addition to instantiability, MMPL validation may require stronger correctness conditions, e.g., discovering some/all meta-models without instances (Analysis 4). A non-instantiable meta-model may indicate that the configuration used to create it has incompatible features.

**Example.** All 16 total configurations in our example yield instantiable meta-models. This means, among other things, that there are Petri nets which are both StateMachines and MarkedGraphs, and there are no incompatible features.

Even if a meta-model has instances, it is reasonable to require that some of them combine objects coming from different features in the configuration. Hence, Analysis 5 relies on feature-exercising instance generation to produce models with instances of all elements activated by the given configuration, returning those configurations where no such models exist. This typically reveals a feature conflict.

**Example.** Fig. 6 illustrates the subtle problems that Analysis 5 can detect. A model that exercises the feature Hierarchical must contain hierarchical transitions (i.e., with non-empty collections places or trans) satisfying the invariant isHierarchical (i.e., with empty collections in and out). However, if the configuration also selects the feature StateMachine, the resulting meta-model does not accept models with hierarchical transitions as they violate the cardinality [1..1] of references in and out (see Fig. 6(a)). Thus, no transition in a state-machine net can be hierarchical, as Fig. 6(b) shows, and so the meta-model elements added by the Hierarchical feature are meaningless when StateMachine is selected as well. Hence, there is no synergy between these two features, which is discovered by Analysis 5. This issue can be solved by adding a modifier that sets the *min* cardinality of references in and out to 0 when both StateMachine and Hierarchical are selected.

While Analyses 1–5 leave the property $P$ empty, Analyses 6–12 focus on the satisfiability of properties. This way, depending on the selected analysis options, it is possible to assess whether some/every/no meta-model in the product line (exists$_{MM}$/forAll$_{MM}$/notExists$_{MM}$) has some/every/no instance satisfying a given property (exists$_m$/forAll$_m$/notExists$_m$). Table 1 shows examples of these analyses both in natural language (for comprehension) and in OCL (to automate validation).

**Example.** To check whether all meta-models in the MMPL admit models with more transitions than places, we have to select the analysis options forAll$_{MM}$ and exists$_m$, and define the OCL property Transition.**allInstances**()→**size**() > Place.**allInstances**()→**size**(). The analysis assesses that this is not the case, as the configurations that select both features StateMachine and MarkedGraph do not admit more transitions than places.

Overall, our classification permits defining different kinds of instance properties applicable at MMPLs, and includes new specification options such as mixed properties with no counterpart in the analysis of plain meta-models.

## 4 Analysing Instantiability Properties

As explained in Section 3.1, checking a property on the instances of a meta-model amounts to finding a model that

TABLE 1
Examples of instance property analyses for MMPLs. Total configurations only show the features in $\rho^+$ (i.e., with value *true*).

| | Analysis options (cf. Fig. 5) | Analysis | Example (P=property; R=result) |
|---|---|---|---|
| 1 | one, exists$_m$, global, config | Find 1 total configuration that produces an instantiable MM | **R:** A total configuration: ⟨PetriNets, Tokens, Object, Hierarchical⟩ |
| 2 | one, exists$_m$, global, model | Find 1 model of 1 instantiable MM | **R:** A model with a PetriNet and a Place objects |
| 3 | all, exists$_m$, global, model | Find 1 model of each instantiable MM | **R:** Sixteen example models are populated |
| 4 | all, notExists$_m$, global, config | Find all total configurations that produce a non-instantiable MM | **R:** Empty set (all configurations are instantiable) |
| 5 | all, notExists$_m$, global, feature exercising, config | Find all total configurations that produce a MM with no instances using every MM element present due to feature selections | **R:** Four total configurations: ⟨PetriNets, Tokens, Simple, StateMachine, Hierarchical⟩, ⟨PetriNets, Tokens, Object, StateMachine, Hierarchical⟩, ⟨PetriNets, Tokens, Simple, StateMachine, MarkedGraph, Hierarchical⟩, ⟨PetriNets, Tokens, Object, StateMachine, MarkedGraph, Hierarchical⟩ |
| 6 | exists$_m$, forAll$_{MM}$, ... | Check if a property is satisfied by some model of every meta-model in the MMPL | **P:** all meta-models admit models with more transitions than places; **R:** *false* (StateMachines that are MarkedGraphs do not) Transition.**allInstances**()→**size**() > Place.**allInstances**()→**size**() |
| 7 | exists$_m$, exists$_{MM}$, ... | Check if a property is satisfied by some model of some meta-model in the MMPL | **P:** some meta-models admit models with a start place; **R:** *true* Place.**allInstances**()→**exists**(p \| Transition.**allInstances**()→**forAll**(t \| t.out→**excludes**(p))) |
| 8 | forAll$_m$, forAll$_{MM}$, ... | Check if a property is satisfied by every model of every meta-model in the MMPL | **P:** all models have at least one place; **R:** *false* Place.**allInstances**()→**notEmpty**() |
| 9 | forAll$_m$, exists$_{MM}$, ... | Check if a property is satisfied by every model of at least one (arbitrary) meta-model in the MMPL | **P:** some meta-models only admit models with more places than transitions; **R:** *false* Place.**allInstances**()→**size**() > Transition.**allInstances**()→**size**() |
| 10 | notExists$_m$, forAll$_{MM}$, ... | Check if no model of the MMPL satisfies a property | **P:** no model has isolated transitions; **R:** *false* Transition.**allInstances**()→**exists**(t \| t.in→**isEmpty**() **and** t.out→**isEmpty**()) |
| 11 | notExists$_m$, exists$_{MM}$, ... | Check if at least one (arbitrary) meta-model of the MMPL has no model satisfying a property | **P:** some meta-models do not accept models with isolated transitions; **R:** *true* (e.g., for StateMachines) Transition.**allInstances**()→**exists**(t \| t.in→**isEmpty**() **and** t.out→**isEmpty**()) |
| 12 | mixed, ... | Check a property mixing meta-model elements and features (mixed property) | **P:** transitions with one input are only on StateMachines; **R:** *false* Transition.**allInstances**()→**forAll**(t \| t.in→**size**()=1) **implies** StateMachine |

conforms to the meta-model and satisfies the prop
search is typically performed using model finders.
to analyse an instantiability property on an MMPL
perform model finding over each meta-model in th
However, this implies solving a model finding pr
meta-model, which can be time-consuming as th
of meta-models in an MMPL may grow exponenti

For this reason, we lift the search over an exte
sion of the *150MM* that contains the meta-model
of every variant and emulates the feature confi
PCs and modifiers using invariants. We call this m
*feature explicit meta-model* (FEMM). This way, the
of checking a property on some/every meta-mo
MMPL is recasted as finding one or more instan
FEMM. By doing so, we reduce the number of mod
problems to solve, which otherwise can be ex
Fig. 7 shows the workflow to perform the analy
duced in Section 3.2.

In Step 1, we *compile* the *150MM*, the featu
and the property of interest into an FEMM. Thi
the *150MM* with invariants that emulate the ser
the PCs and modifiers, an extra class FMC with
attribute for each feature in the feature model, an
stating the allowed configurations, and the property to check. Fig. 8 shows the FEMM for the running example whose construction is explained below.

In Step 2, we use a model finder to search for an instance of the FEMM which exemplifies (or falsifies if no model is found) the existence of models satisfying the property. If an instance of the FEMM is found, then it contains both a model satisfying the property and an object of type FMC

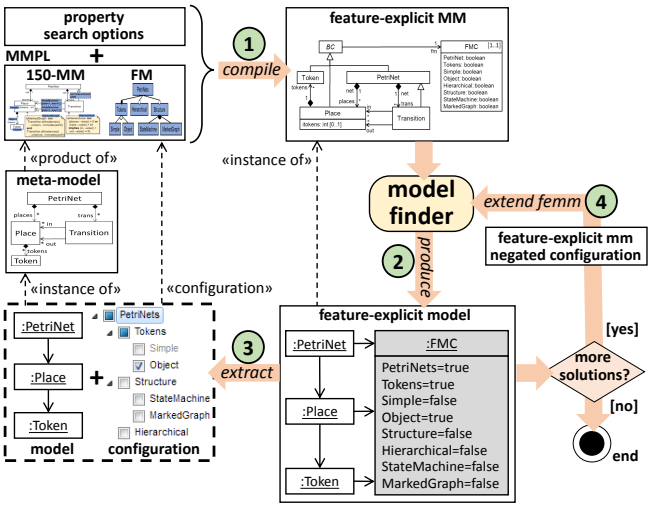



Fig. 7. Workflow of instance property analysis.

reporting a feature configuration. In Step 3, we extract the configuration and the model as two separate artefacts. If required, the configuration can be applied to the MMPL to produce the meta-model of which the model is an instance.

Finally, if the user wants to identify further meta-models satisfying the property, Step 4 extends the FEMM with an invariant requiring a configuration different from the ones already found. For this purpose, the invariant disallows objects of type FMC to take the same attribute values as the configurations found so far. Then a new search is performed.

Next, we explain the steps to build the FEMM (cf. List-

```
1  buildFeatureExplicitMM (MMPL : in, FEMM : out)
2    create FEMM = new meta-model
3    add classes, fields, invariants, inheritance in 150MM to FEMM
4
5    // 1) embed feature model in feature-explicit meta-model
6    create FMC = new concrete class
7    create BC = new abstract class
8    create fm = new reference from BC to FMC, with cardinality [1,1]
9    forall feature f in FM do
10     add boolean attribute f to FMC, with cardinality [1,1]
11   add invariant to FMC = formula in FM
12   add BC as superclass to all classes with no parent class
13   add FMC, BC and fm to FEMM
14
15   // 2) emulate presence conditions of classes and invariants
16   forall class c with presence condition pc do
17     add invariant to FMC = "not pc implies size(c) = 0"
18   forall invariant wc with presence condition pc do
19     set wc = "pc implies wc"
20
21   // 3) emulate conditions of fields (max modifiers omitted for brevity)
22   forall field f with presence condition pc, or
23                    with min modifier m and modifier condition mc do
24     set cond = "size(f) ⩾ f.min"
25     if mc is defined then
26       set cond = "if mc then size(f) ⩾ m else size(f) ⩾ f.min endif"
27     if pc is defined then
28       add invariant to owner(f) = "if pc=false then size(f) = 0
29                                    else " + cond + " endif"
30       set f.min = 0
31     else
32       add invariant to owner(f) = cond
33       set f.min = minimum(f.min, m)
34
35   // 4) emulate inheritance modifiers (add modifiers omitted for brevity)
36   forall inheritance link inh with del modifier condition mc do
37     forall field f inherited through inh do
38       add invariant to subclass = "mc implies size(f) = 0"
39     forall reference r to the superclass or an ancestor do
40       add invariant to owner(r) = "mc implies
41                                    not r.exists(o|o.oclIsKindOf(subclass))"
42     forall invariant wc with pres. cond. pc inherited through inh do
43       add invariant to subclass = ''pc and not mc implies wc''
44       set wc = ''not self.oclIsKindOf(subclass) implies wc''
45
46   // 5) exercise features (optional, add modifiers omitted for brevity)
47   forall class c with presence condition pc do
48     add invariant to FMC = "pc implies size(c) > 0"
49   forall field f with presence condition or modifier pc do
50     add invariant to FMC = "pc implies size(o.f) > 0" for some o
51   forall inheritance link inh with del modifier condition mc do
52     forall field f inherited through inh do
53       add invariant to FMC = "mc=false implies size(o.f) > 0"
54                              for some o
```

Listing 1. Simplified algorithm to build the FEMM.

ing 1 and Fig. 8) and perform the different analysis types.

**1) Embedding the feature model.** The FEMM is structurally similar to the *150MM*, as it holds the elements in every possible meta-model variant (lines 2–3 in Listing 1). Moreover, it embeds the feature model (lines 6–13) represented as a class FMC with a boolean attribute for each feature, and an invariant that corresponds to the propositional formula governing the allowed configurations (see invariant $\phi_{MMPL}$ of class FMC in Fig. 8). To make the class FMC accessible from any other, we create a base class BC for the rest of classes, and add a reference from it to FMC. This way, every class has access to the value of the attributes in FMC, i.e., to the feature configuration. As the cardinality inscription in class FMC shows, we require exactly one instance of this class in the model finding process.

**2) Emulating the PCs of classes and invariants.** PCs are emulated by extra invariants in the FEMM, as shown in lines 16–19 of Listing 1.

A PC in a class is converted into an invariant of FMC ensuring that there are no instances of the class when the PC is not *true*. The class FMC is the appropriate context for the invariant, as we require exactly one object of this class. As an example, class FMC in Fig. 8 declares the invariant wc-Token to represent the PC of class Token. The invariant requires zero objects of type Token when not Object is *true*. This captures the fact that the Token class is not included in meta-models of configurations where Object is not selected. If a PC affects an invariant, the latter is rewritten so that it is only checked when the PC is met. For example, invariant isMarkedGraph in class Place is rewritten to be applicable only when MarkedGraph is *true*.

**3) Emulating the PCs and modifiers of fields.** Lines 22–33 of Listing 1 handle the PCs and modifiers of fields. A PC in a field is represented as an invariant in the field's owner class which ensures that the field is empty (in case of references) or undefined (for attributes) when the PC is *false*. Moreover, the *min* cardinality of the field is set to 0, as the field could never be empty otherwise. As an example, the invariant wc-itokens in class Place is derived from the PC of Place.itokens. The invariant requires the attribute to be undefined when the PC is not satisfied, but to have a value (because of the original *min* cardinality 1) otherwise. The *min* cardinality of the attribute, which was 1 in the *150MM*, becomes 0.

If a field has cardinality modifiers then the created invariant also ensures the satisfaction of the modifier cardinality when the modifier condition is *true*, or the cardinality specified in the *150MM* otherwise. For example, invariants wc-in-min and wc-in-max in class Transition are derived from the *min* and *max* modifiers in reference Transition.in. In addition, the *min* cardinality of the field is set to the minimum between its original value, the values given by its *min* cardinality modifiers, and 0 if the field has a PC; while its *max* cardinality becomes the maximum between its original value and its *max* cardinality modifiers. In the example, Transition.in does not change its cardinality as it originally had the minimum and maximum possible values ([0..*]).

**4) Emulating the inheritance modifiers.** Inheritance modifiers are handled in lines 36–44 of Listing 1. A modifier deleting (resp. adding) an inheritance link is translated into an invariant in the subclass requiring that any field inherited through the inheritance link has no value when the modifier condition is *true* (resp. *false*). As an example, invariant wc-del-inh in class Transition is derived from the inheritance modifier in the *150MM*. In both cases, the inheritance link is added to the FEMM. This limits the applicability of our approach to the cases where the FEMM has no inheritance cycles, condition that can be detected a-priori statically. Moreover, for each incoming reference to the superclass or an ancestor, additional invariants check that the reference does not contain instances of the subclass when the modifier condition is *true* (resp. *false*). Invariants wc-net in classes Place and Transition are generated for this reason, each one coming from an incoming reference net to the superclass PetriNet. Finally, any invariant defined

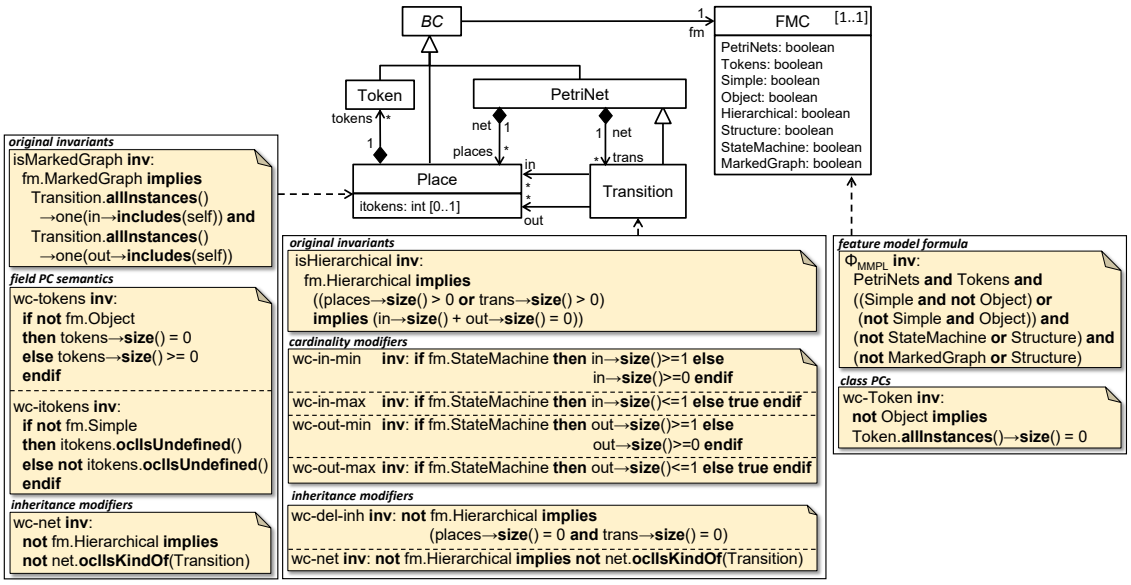



Fig. 8. FEMM derived from the MMPL in Figs. 2 and 3.

in the superclass or an ancestor is rewritten to avoid its application on the subclass instances, and a copy of the invariant is added to the subclass, modified so that it is only checked when the modifier condition is *false* (resp. *true*).

**5) Exercising features (optional).** Exercising features allows illustrating the meta-model elements specific to the features selected in a configuration. For this purpose, when choosing this option, class FMC is added invariants enforcing the existence of instances of the classes and fields activated by the selected features (lines 47–54 in Listing 1). A PC annotating a class becomes an invariant requiring at least one object of the class when the PC is *true* (so that the class is exercised). A PC or a modifier on a field is translated into an invariant requiring the field to have a non-empty value in some object when the presence or modifier condition is *true* (so that the field is exercised). A modifier deleting (resp. adding) an inheritance link is translated into an invariant requiring that, when the modifier condition is *false* (resp. *true*), the inherited fields have a non-empty value in some object. Fig. 8 omits these invariants for clarity.

For example, when exercising features, the invariant Object **implies** Token.**allInstances**()→**size**()>0 is added to class FMC. As a consequence, any example model of a configuration that selects the feature Object needs to have Token objects.

**6) Embedding the property.** The property to analyse and the configuration scope (when partial) are added as invariants to class FMC. Even if the property is mixed, it can be embedded without changes because the class FMC defines the feature values as attributes, so they are accessible from the added invariant. If the aim is checking whether the property is satisfiable by all instances of a meta-model, then the property is negated.

As an example, to embed a property applicable to configurations where the feature StateMachine is selected, we add the invariant self.StateMachine to class FMC.

**7) Analysing the property.** Once the FEMM has been constructed, we look for an instance of it using a model finder. We require this instance to contain exactly one object of type FMC, which will hold a valid total configuration.

To analyse whether some meta-model in the MMPL has instances satisfying a property (feature $\text{exists}_{MM}$ in Fig. 5), we invoke the finder with the FEMM as input. If a result is found, it becomes a witness to the satisfiability of the property by some instance of a meta-model of the MMPL. Since the finder returns an instance of the FEMM, we need to extract the actual model and configuration (i.e., the attribute values in the FMC object) from it. Next, the model's meta-model is produced from the obtained configuration. Altogether, this process yields a configuration, the corresponding meta-model, and an instance model satisfying the property. Finding the configuration of another solution meta-model is done by adding an invariant that disallows the found configurations to class FMC, and invoking the finder again. This process can be iterated until the finder returns no further configurations. In Section 7.2, we evaluate the performance of this iterative search according to different strategies.

**Example.** Fig. 7 illustrates the analysis process: the finder outputs an instance of the FEMM (lower-right), from which a model, a configuration and a meta-model are extracted (lower-left). To obtain another solution meta-model, we disallow the found configuration by adding to the FEMM the invariant **not** (PetriNets **and** Tokens **and not** Simple **and** Object **and not** Structure **and not** Hierarchical **and not** StateMachine **and not** MarkedGraph). Then we perform a new model search.

To assess that no meta-model in the MMPL has instances that satisfy a property (feature $\text{notExists}_{MM}$), the model finder is invoked iteratively, each time excluding the previously found configurations. When the finder does not find new results, the analysis returns the remaining valid configurations not found by the solver. These identify the meta-models with no instances satisfying the property.

Finally, assessing whether all meta-models in an MMPL have instances satisfying a property (feature forAll$_{MM}$) is similar to checking that no meta-model has such instances. The differences are that the property is negated, and the remaining configurations identify the meta-models with all their instances satisfying the property.

The described lifted analysis permits finding a total configuration $\rho$ and a model $M$ satisfying a property $P$ efficiently by performing a single model search. Otherwise, we would need to enumerate and analyse each meta-model $MM_\rho \in Prod(MMPL)$ until finding one with instances satisfying $P$. In the worst case, if no model satisfies $P$, our lifted analysis finishes after one call to a model finder, while the enumerative approach requires as many calls as there are meta-models in the MMPL.

However, to get *all* meta-models $MM_\rho \in Prod(MMPL)$ with instances satisfying $P$, our approach requires as many calls to a model finder as there are satisfying meta-models. If all meta-models satisfy $P$, then it requires as many calls as there are meta-models in the MMPL. In this case, our lifted analysis may be less efficient than the enumerative approach, as model finding is generally more expensive over the FEMM than over a meta-model of the MMPL. Hence, the next section presents the main contribution of the paper, which is an extension of the presented analysis to improve its efficiency to deal with this latter case. The idea is allowing partial configurations as results of the model search, accounting for several total configurations. This permits reducing number of model finder calls.

**input :** $MMPL = (FM = (F, \phi_{MMPL}), MM, \Phi, M_C, M_I)$, $\tilde{\rho} = \langle \rho^+, \rho^- \rangle \in \tilde{P}(FM)$
**output:** $MMPL^{\tilde{\rho}} = (FM^{\tilde{\rho}}, MM^{\tilde{\rho}}, \Phi^{\tilde{\rho}}, M_C^{\tilde{\rho}}, M_I^{\tilde{\rho}})$

1 **set** $FM^{\tilde{\rho}} = (F \setminus (\rho^+ \cup \rho^-), \phi^{\tilde{\rho}}_{MMPL})$
2 **forall** $x \in MM_C \cup MM_{FI} \cup MM_{WC}$ **do**
3  **if** $\Phi^{\tilde{\rho}}_x \ncong$ *false* **then add** $x$ **to** $MM^{\tilde{\rho}}$
4 **forall** $(C_{sub}, C_{super}) \in MM_I$ with no modifier **do**
5  **if** $\{C_{sub}, C_{super}\} \subseteq C^{\tilde{\rho}}$ **then**
6  **add** $(C_{sub}, C_{super})$ **to** relation $I^{\tilde{\rho}}$
7 **forall** $\langle f, m, \Phi_{min} \rangle \in \mu_{min}$ **do**
8  **if** $\Phi^{\tilde{\rho}}_{min} \cong$ *true* **then set** *f.min* = $m$
9 **forall** $\langle f, m, \Phi_{max} \rangle \in \mu_{max}$ **do**
10  **if** $\Phi^{\tilde{\rho}}_{max} \cong$ *true* **then set** *f.max* = $m$
11 **forall** $\langle C_{sub}, C_{super}, \Phi_{add} \rangle \in \mu_{add}$ **do**
12  **if** $\Phi^{\tilde{\rho}}_{add} \ncong$ *false* **then add** $(C_{sub}, C_{super})$ **to** relation $I^{\tilde{\rho}}$
13 **forall** $\langle C_{sub}, C_{super}, \Phi_{del} \rangle \in \mu_{del}$ **do**
14  **if** $\Phi^{\tilde{\rho}}_{del} \ncong$ *true* **then add** $(C_{sub}, C_{super})$ **to** relation $I^{\tilde{\rho}}$
15 **set** $\Phi^{\tilde{\rho}}_X = \{\langle x, \Phi^{\tilde{\rho}}_x \rangle \mid \langle x, \Phi_x \rangle \in \Phi_X\}$ for $X \in \{C, FI, WC\}$
16 **set** $\mu^{\tilde{\rho}}_M = \{\langle f, m, \Phi^{\tilde{\rho}}_M \rangle \mid \langle f, m, \Phi_M \rangle \in \mu_M \wedge \Phi^{\tilde{\rho}}_M \ncong$ *false*
17  $\wedge\ \Phi^{\tilde{\rho}}_M \ncong$ *true*$\}$ for $M \in \{min, max\}$
18 **set** $\mu^{\tilde{\rho}}_N = \{\langle C_{sub}, C_{super}, \Phi^{\tilde{\rho}}_N \rangle \mid \langle C_{sub}, C_{super}, \Phi_N \rangle \in \mu_N$
19  $\wedge\ \Phi^{\tilde{\rho}}_N \ncong$ *false* $\wedge\ \Phi^{\tilde{\rho}}_N \ncong$ *true*$\}$ for $N \in \{add, del\}$

**Algorithm 2:** MMPL concretization. Notation is from Defs. 2, 3 and 4.

## 5 Analysis via Partial Configurations

A *partial configuration* is a configuration where some feature values are undefined. Thus, it represents *a set* of total configurations: those that result from substituting the undefined values with *true* or *false*, whenever the feature model formula allows it. Formally, $equiv(\tilde{\rho}) = \{\rho_i \in P(FM) \mid \rho^+ \subseteq \rho_i^+ \wedge \rho^- \subseteq \rho_i^-\}$ is the set of total configurations $\rho_i$ equivalent to the partial configuration $\tilde{\rho}$, with $\rho^+$ and $\rho^-$ the sets of features from $\tilde{\rho}$ set to *true* and *false* (cf. Def. 4).

We improve the efficiency of our lifted analysis by allowing reasoning over partial configurations. Theoretically, we do so by defining a *concretization* operation which, given an MMPL and a partial configuration, resolves any presence and modifier conditions that are not undefined under the given configuration, updating the *150MM* accordingly. Then, we do model finding on the modified MMPL to obtain a witness model. This model can be sliced by an operation similar to a pullback [34] to obtain a valid instance of each meta-model in the modified MMPL.

We implement this approach by enabling the model finder to return, in addition to a witness model of the satisfiability of a property, a partial configuration instead of a total one. Then, for each total configuration $\rho_i$ derivable from the partial one, we build a model $M$ that is an instance of the meta-model $MM_{\rho_i}$. We propose two approaches, called *hard* and *soft*, to achieve this behaviour, as well as heuristics to guide the model search.

Interestingly, the new version of our analysis over partial configurations does not impose any additional restrictions compared to the analysis over total configurations, and the application scope remains the same.

Next, we define product line concretizations via partial configurations in Section 5.1, and enable model finding with partial configurations in Section 5.2.

### 5.1 MMPL Concretizations via Partial Configurations

We define a concretization operation which, given an MMPL *MMPL* and a (partial) configuration $\tilde{\rho}$, produces an MMPL $MMPL^{\tilde{\rho}}$ resulting from the partial evaluation of its formulas. Algorithm 2 describes this operation. Given a formula $\Phi$, the algorithm uses $\Phi^{\tilde{\rho}}$ to denote the substitution $\Phi[true/\rho^+, false/\rho^-]$ that replaces all variables that refer to features in $\rho^+$ by *true*, and to features in $\rho^-$ by *false*. In addition, it uses $\cong$ to check whether such a substitution makes a formula *true* or *false*. Line 1 adds to the feature model $FM^{\tilde{\rho}}$ all undefined features, and partially evaluates $\phi_{MMPL}$. Lines 2–3 add to the *150MM* of $MMPL^{\tilde{\rho}}$ all meta-model elements whose PC is not *false* under the given substitution. Lines 4–6 add the inheritance relations with no modifiers, if they relate classes added in the previous step. Lines 7–10 modify the field cardinalities according to the *min* and *max* modifiers, whenever their conditions evaluate to *true* under the given substitution. Similarly, lines 11–14 add all inheritance relations with non-*false add* modifiers, or non-*true del* modifiers. Finally, lines 15–19 build all presence and modifier conditions in $MMPL^{\tilde{\rho}}$.

**Example.** Fig. 9 concretizes the MMPL *MMPL* through the partial configuration $\tilde{\rho} = \langle \{\text{PetriNets}, \text{Structure}, \text{StateMachine}\}, \{\text{Hierarchical}, \text{MarkedGraph}\} \rangle$ yielding $MMPL^{\tilde{\rho}}$. The resulting *150MM* includes only the meta-model classes, fields and invariants whose PC is not *false* when PetriNets, Structure and StateMachine are substituted by *true*, and Hierarchical and MarkedGraph by *false*. The inheritance relation from Transition to PetriNet is deleted because the condition of the *del* modifier

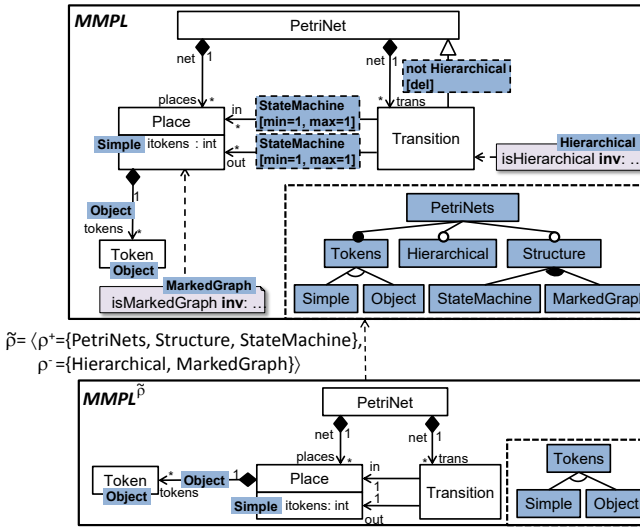



Fig. 9. An example MMPL concretization.

becomes *true* when substituting Hierarchical by *false*. The cardinality of references in and out becomes 1 as their cardinality modifiers are *true* under the configuration $\tilde{\rho}$. Finally, in the presence and modifier conditions, the feature names are substituted by their values when the features are defined, and the conditions are deleted when they become *true* or *false* (e.g., the cardinality modifiers for in and out).

A product line $MMPL^{\tilde{\rho}}$ is called *ground* if $F^{\tilde{\rho}} = \emptyset$ (i.e., it has an empty set of features) and thus it is equivalent to a regular meta-model. Hence, we can recast the set of product meta-models $Prod(MMPL)$ as the set of ground MMPL concretizations using all possible non-partial configurations. Lemma 1 captures this intuition.

**Lemma 1** (Ground concretizations are products)**.** Given a product line *MMPL* and a total configuration $\rho$, $MMPL^{\rho} = (FM^{\rho}, MM^{\rho}, \Phi^{\rho}, M_C^{\rho}, M_I^{\rho})$ is a ground MMPL, and $MM^{\rho} \in Prod(MMPL)$.

*Proof.* To prove the lemma, we have to show that (i) concretizations by total configurations lead to ground MMPLs, and (ii) if $MMPL^{\rho}$ is a concretization of *MMPL*, the *150MM* of $MMPL^{\rho}$ is a product of *MMPL*.

We prove (i) by noting that a total configuration $\rho$ has no undefined features, i.e., $\rho^u = \emptyset$. Hence, in line 1 of Algorithm 2, $F \setminus (\rho^+ \cup \rho^-) = \rho^u = \emptyset$, and $\phi^{\rho}_{MMPL} = true$.

For (ii), as $\rho$ is total, any presence or modifier condition $\Phi^{\rho}$ evaluates to *true* or *false*. Hence, in line 3 of Algorithm 2, checking $\Phi^{\tilde{\rho}}_x \not\cong false$ is equivalent to checking $\Phi^{\tilde{\rho}}_x$, and as in Algorithm 1, the elements whose PC is *true* are kept, and the inheritance relations between the selected classes are maintained. The checks in lines 8, 10, 12 and 14 of Algorithm 2 ($\Phi^{\tilde{\rho}}_{min} \cong true$, $\Phi^{\tilde{\rho}}_{max} \cong true$, $\Phi^{\tilde{\rho}}_{add} \not\cong false$, $\Phi^{\tilde{\rho}}_{del} \not\cong true$) are equivalent to those in lines 7, 9, 11 and 14 of Algorithm 1 ($\Phi^{\tilde{\rho}}_{min}$, $\Phi^{\tilde{\rho}}_{max}$, $\Phi^{\tilde{\rho}}_{add}$, $\neg\Phi^{\tilde{\rho}}_{del}$), respectively, and so both meta-models are equal. □

The next step is to lift the conformance between a model and a meta-model to the MMPL level. For this purpose, we first define a simple notion of *a model*.

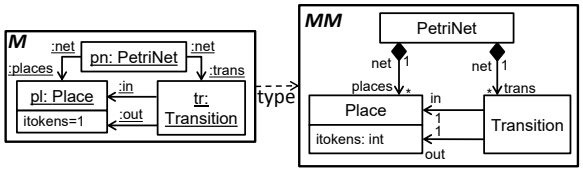



Fig. 10. Model type example.

**Def. 6** (Model)**.** A *model* is a tuple $M = (O, SL)$, where $O$ is a set of objects and $SL$ is a set of slots and links, each owned by exactly one object.

Slots and links represent instances of attributes and references, respectively. We sometimes use a function $owner_{SL} : SL \to O$ to indicate the owner object of slots and links, the way we do for fields in meta-models. Given a slot or a link $sl \in SL$, we use $target(sl)$ to refer to the target object of the link, or the value of the slot. Next, we define the type of a model as a function from the model to a meta-model.

**Def. 7** (Model type)**.** Given a meta-model $MM = (C, FI, WC, I)$ and a model $M = (O, SL)$, we define the *model type* of $M$ as the function $type : M \to MM$, which is a tuple $type = \langle type_O, type_{SL} \rangle$ consisting of two functions:

- $type_O : O \to C$ mapping objects to non-abstract classes;
- $type_{SL} : SL \to FI$ mapping slots to attributes and links to references, keeping their source/target compatibility:
  1) $type_O \circ owner_{SL} \subseteq owner^*_{FI} \circ type_{SL}$, and
  2) for each link $l \in SL \cdot target(type_{SL}(l)) \in ancs(type_O(target(l)))$.

As in most type systems, $type_O$ does not allow typing objects by abstract classes. The first condition for $type_{SL}$ ensures that the type of a link (or slot) is owned or inherited by the type of the link's owner (recall that $owner^*_{FI}$ is the relation representing all classes that define or inherit a field). The second condition requires each link target to be compatible with the target class of the link type (recall that *ancs* is the reflexive-transitive closure of $I$). We deliberately leave out a similar compatibility condition for slots, as it is not relevant for our theory.

**Example.** Fig. 10 shows a model type example $type = \langle type_O = \{(\text{pn}, \text{PetriNet}), (\text{pl}, \text{Place}), (\text{tr}, \text{Transition})\}, type_{SL} = \{(\text{pn.places}, \text{PetriNet.places}), (\text{pn.trans}, \text{PetriNet.trans}), (\text{pl.itokens}, \text{Place.itokens}), (\text{tr.in}, \text{Transition.in}), (\text{tr.out}, \text{Transition.out})\}\rangle$.

Next, we define the notion of model conformance. Given a model typed by a meta-model, we say that the model conforms to the meta-model if every slot and link in the model respects the cardinality of its type (an attribute or a reference, respectively), and every model object satisfies the invariants defined by the object's type and supertypes.

**Def. 8** (Conformance)**.** Given a model type $type : M \to MM$, $M$ *conforms to* $MM$ via *type*, written $M \models_{type} MM$, iff:

1) slots and links respect the cardinality of their type: $\forall o \in O, \forall f \in FI \; s.t. \; type_O(o) \in owner^*_{FI}(f) \cdot min \leq |\{sl \mid type_{SL}(sl) = f \wedge owner_{SL}(sl) = o\}| \leq max$, with $[min, max]$ the cardinality of $f$, and
2) each object $o \in O$ satisfies the invariants defined by $type_O(o)$, and its supertypes.

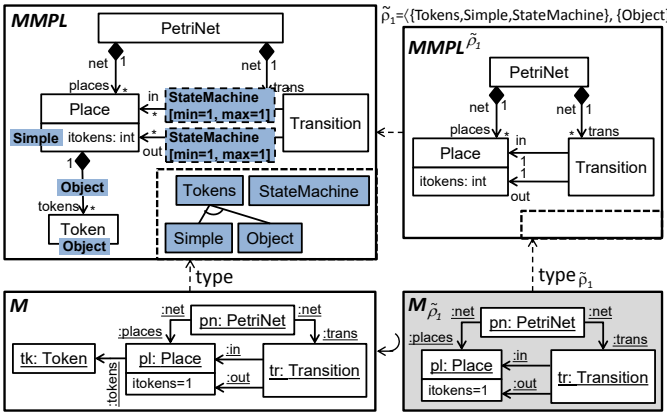



Fig. 11. Model slicing and conformance example.

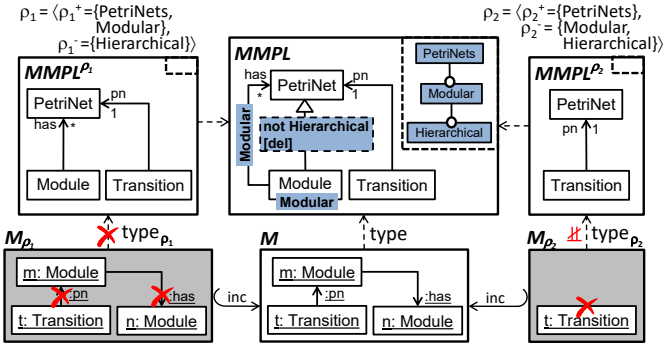



Fig. 12. Slice with invalid model type (left), and non-conformance (right).

**Example.** In Fig. 10, we have that $M \models_{type} MM$ because all slots and links of pn, pl and tr satisfy their cardinality constraints, and *MM* has no invariants.

Next, we lift our notion of conformance to the product line level. This way, we say that a model $M$ conforms to an MMPL if it conforms to *all* product meta-models represented by the MMPL. We call this *universal conformance*.

**Def. 9** (Universal conformance)**.** Given a product line $MMPL = (FM, MM, \Phi, M_C, M_I)$ and a model type $type\colon M \to MM$, *$M$ conforms to MMPL* via *type*, written $M \models_{type} MMPL$, iff:

1) slots and links respect the cardinality and modifiers of their type: $\forall o \in O, \forall f \in FI\ s.t.\ type_O(o) \in owner^*_{FI}(f) \cdot lmin \leq |\{sl \mid type_{SL}(sl) = f \wedge owner_{SL}(sl) = o\}| \leq lmax$, with *lmin* being the highest of all *min* cardinalities of $f$, and *lmax* being the lowest of its *max* cardinalities, and
2) each object $o \in O$ satisfies the invariants defined by $type_O(o)$, and its supertypes.

**Example.** On the left hand side of Fig. 11, $M \models_{type} MMPL$. This is the case as the function *type* is a model type (cf. Def. 7), and tr.in and tr.out satisfy both the cardinality and modifiers of their reference types. If *MMPL* had invariants with PC, the model $M$ should satisfy them as well.

Next, we define an operation *slice* which slices a model $M$ conforming to an MMPL so that the resulting model slice is typed by a concretization of the MMPL (i.e., it contains the instances of the meta-model elements that belong to the concretization).

**Def. 10** (Model slice)**.** Given a product line *MMPL*, a partial configuration $\tilde{\rho}$, a concretization $MMPL^{\tilde{\rho}}$, and a model $M = (O, SL)$ s.t. $M \models_{type} MMPL$, a *slice* of $M$ w.r.t. $MMPL^{\tilde{\rho}}$, written $M|_{MMPL^{\tilde{\rho}}} = (O^{\tilde{\rho}}, SL^{\tilde{\rho}})$, is comprised of:

- the objects typed by classes of $MM^{\tilde{\rho}}$: $O^{\tilde{\rho}} = \{o \in O \mid type_O(o) \in MM^{\tilde{\rho}}_C\}$, and
- the slots and links typed by fields of $MM^{\tilde{\rho}}$: $SL^{\tilde{\rho}} = \{sl \in SL \mid type_{SL}(sl) \in MM^{\tilde{\rho}}_{FI} \wedge owner_{SL}(sl) \in O^{\tilde{\rho}} \wedge target(sl) \in O^{\tilde{\rho}}\}$.

*Remark.* This operation also defines the restriction of $M$ w.r.t. a meta-model $MM_\rho \in Prod(MMPL)$ since ground concretizations are products.

**Example.** Fig. 11 shows a model slice $M_{\tilde{\rho}_1}$ derived from a model $M \models_{type} MMPL$ and a partial configuration $\tilde{\rho}_1$ (shown on top). $M_{\tilde{\rho}_1}$ is the restriction of $M$ to the elements in $MM^{\tilde{\rho}_1}$. This way, according to Def. 10, the slice contains the objects of $M$ typed by classes of $MM^{\tilde{\rho}_1}$ (set $O^{\tilde{\rho}}$ in the definition) as well as the slots and links typed by fields of $MM^{\tilde{\rho}_1}$ (set $SL^{\tilde{\rho}}$ in the definition).

Given a model type $M \models_{type} MMPL$ and a concretization $MMPL^{\tilde{\rho}}$, the restriction of *type* to $MM^{\tilde{\rho}}$, written $type_{\tilde{\rho}}\colon M|_{MMPL^{\tilde{\rho}}} \to MM^{\tilde{\rho}}$, may not fulfil the conditions for a model type (cf. Def. 7). The left hand side of Fig. 12 shows an example. $M_{\rho_1}$ is a slice of $M$ built using the configuration $\rho_1$. However, $type_{\rho_1}$ is not a model type: since $\rho_1$ selects feature Modular but not Hierarchical, links of type Module.has can no longer contain Module objects, and Module objects can no longer be assigned to links of type Transition.pn. Next, we introduce a notion of monotonicity for references to ensure a correct type for model slices. A reference may be non-monotonic if the set of compatible classes with its source and target classes can change due to inheritance modifiers.

**Def. 11** (Non-monotonic reference)**.** Let $r$ be a reference with PC $\Phi_r$, $owner_{FI}(r) = C_s$ and $target(r) = C_t$. Let $\Phi_{C'_x}$ be the condition, induced by inheritance modifiers, for a class $C'_x$ with $C_x \in ancs(C'_x)$ to be no longer a subclass of $C_x$ (for $x = \{s, t\}$). We say that $r$ is *non-monotonic* if $\Phi_{C'_s} \not\Rightarrow \neg\Phi_r$ or $\Phi_{C'_t} \not\Rightarrow \neg\Phi_r$.

**Example.** In Fig. 12, reference Module.has is non-monotonic: when **not** Hierarchical is *true*, Module is not compatible with PetriNet. This makes models where some has link contains Module objects (e.g., $M_{\rho_1}$) ill-typed. The reference would be monotonic if it had PC Hierarchical **and** Modular. Similarly, Transition.pn is non-monotonic: when **not** Hierarchical is *true*, class Module becomes incompatible with the reference target. Hence, models in which some Module object receives a link pn (e.g., $M_{\rho_1}$) are ill-typed. In contrast, all references in Fig. 11 are monotonic so slicing yields a valid model type.

Given a model conformant to an MMPL without non-monotonic references, any slice of the model built as described in Def. 10 is well-typed. The next theorem captures this fact.

**Thrm. 1** (Slicing yields model types)**.** Given a product line *MMPL*, a partial configuration $\tilde{\rho}$, and a model $M$ s.t.

$M \models_{type} MMPL$, $type_{\tilde{\rho}}\colon M|_{MMPL^{\tilde{\rho}}} \rightarrow MM^{\tilde{\rho}}$ is a model type iff *MMPL* does not contain non-monotonic references.

*Proof.* We start by assuming that *MMPL* has only monotonic references, and show that $type_{\tilde{\rho}}$ is a model type according to Def. 7. First, no object $o$ can have an abstract type through $type_{\tilde{\rho}}$, since $type$ is a model type. Second, since every reference is monotonic, Condition 1 for links in Def. 7 holds: given any reference $r$, for each subclass $C'_s$ of $owner_{FI}(r)$, we have $\Phi_{C'_s} \Rightarrow \neg\Phi_r$, with $\Phi_{C'_s}$ as in Def. 11. Then, in any slice that contains $r$, $\Phi_r$ is *true*, but $\Phi_{C'_s}$ cannot be *true* since $\Phi_{C'_s} \Rightarrow \neg\Phi_r$ and we would have a contradiction. Hence, $C'_s$ is a subclass of $owner_{FI}(r)$ in these slices, and Condition 1 holds for $type_{\tilde{\rho}}$, since it holds for $type$. The same reasoning applies to $target(r)$, so Condition 2 of Def. 7 holds.

Next, we assume that *MMPL* has a non-monotonic reference $r$ due to a subclass $C'_s$ of $owner_{FI}(r)$. Then, Condition 1 for links in Def. 7 will not hold for those slices in which $\Phi_{C'_s}$ is *true* and some $C'_s$ object owns links of type $r$, since $C'_s$ does not inherit $r$ when $\Phi_{C'_s}$ is *true*. A similar reasoning for $target(r)$ causes failure of Condition 2. □

An MMPL without non-monotonic references yields correctly typed model slices, but does not guarantee conformance, as the right side of Fig. 12 shows. Since object m is removed from the slice $M_{\rho_2}$, object t violates the cardinality of reference Transition.pn. The reason for non-conformance is that the constraints affecting Module (the cardinality of Transition.pn) still exist when Modular is *false*. The next definition characterizes this kind of cardinality and well-formedness constraints which we also call *non-monotonic*.

**Def. 12** (Monotonic MMPL)**.** An invariant $wc$ with PC $\Phi_{wc}$ is *non-monotonic* if it affects objects of a class $A$ with PC $\Phi_A$ s.t. $\Phi_{wc} \not\Rightarrow \Phi_A$. A (*min* or *max*) cardinality constraint $d$ of a reference $r$ is *non-monotonic* if:

- $d \notin \{0, \infty\}$, and
- $r$ can contain objects of a class $A$ when $\Phi_A$ is *true*, and $\Phi_d \not\Rightarrow \Phi_A$, where $\Phi_d$ is the condition of the cardinality modifier setting the *min* or *max* cardinality of $r$ to $d$, or $\Phi_r$ if $r$ has no cardinality modifiers.

A product line *MMPL* is *monotonic* if it has no non-monotonic references, invariants or cardinalities.

*Remark.* In the condition for non-monotonic cardinalities, $\Phi_A$ can either be the PC of the class $A$ or the conjunction of the inheritance modifiers of an inheritance path leading to $target(r)$. The case of non-monotonic cardinalities can be reduced to the case of non-monotonic invariants since cardinality constraints can be expressed as invariants. Monotonic references can have non-monotonic cardinality, since non-monotonicity in references is concerned with the modification of inheritance relations, while in cardinalities it is concerned with the deletion of subclasses. Non-monotonic references, cardinalities and invariants are problematic only in those models that have instances of the classes that cause non-monotonicity (e.g., Module in Fig. 12).

**Example.** The *min* cardinality of Transition.pn in Fig. 12 is non-monotonic because *true* $\not\Rightarrow$ Modular $\wedge$ Hierarchical. Hence, it may lead to non-conformant slices such as $M_{\rho_2}$. Should class Transition have the invariant PetriNet.**allInstances**()→**notEmpty**() with PC *true*, it would be non-monotonic because it affects objects of class Module when Modular $\wedge$ Hierarchical is *true*, but this is not implied by *true*.

Next, we show that, given a model $M$, a monotonic product line *MMPL* such that $M \models_{type} MMPL$, and any concretization $MMPL^{\tilde{\rho}}$ of *MMPL*, the slice of $M$ w.r.t. $MMPL^{\tilde{\rho}}$ conforms to $MMPL^{\tilde{\rho}}$.

**Thrm. 2** (Universal conformance yields concretization conformance)**.** Given a monotonic product line *MMPL* and a model $M$ s.t. $M \models_{type} MMPL$, $\forall\tilde{\rho} \in \tilde{P}(FM) \cdot M|_{MMPL^{\tilde{\rho}}} \models_{type_{\tilde{\rho}}} MMPL^{\tilde{\rho}}$.

*Proof.* Given a model $M = (O, SL)$ with $M \models_{type} MMPL$, a concretization $MMPL^{\tilde{\rho}}$ for partial configuration $\tilde{\rho} \in \tilde{P}(FM)$, and the slice $M|_{MMPL^{\tilde{\rho}}} = (O^{\tilde{\rho}}, SL^{\tilde{\rho}})$, we know that the type restriction is a model type by Theorem 1. Hence, we only need to show that the two conditions in Def. 9 hold for $M|_{MMPL^{\tilde{\rho}}} \models_{type_{\tilde{\rho}}} MMPL^{\tilde{\rho}}$.

Condition 1: We exploit the remark after Def. 3 that cardinality modifiers can be expressed as invariants. This reduces Condition 1 to a special case of Condition 2.

Condition 2: Let $wc$ be an invariant in $MMPL^{\tilde{\rho}}$. Then its PC $\Phi_{wc}$ cannot be *false* in $MMPL^{\tilde{\rho}}$. Since $M \models_{type} MMPL$, $M$ satisfies $wc$, and the only way for $M|_{MMPL^{\tilde{\rho}}}$ to violate $wc$ is when the slice deletes an object, link or slot. Suppose that $wc$ is violated due to deleting an object $o \in O \setminus O^{\tilde{\rho}}$ with $type_O(o) = C$, where the PC $\Phi_C$ evaluates to *false* in $\tilde{\rho}$. If $wc$ is violated due to deleting $o$, then $wc$ must affect the class $C$. Since *MMPL* is monotonic, then by Def. 12, $\Phi_{wc} \Rightarrow \Phi_C$. Thus, since $\Phi_C$ is *false*, $\Phi_{wc}$ must be *false*. But this is a contradiction because $wc$ cannot be *false*. Therefore, $wc$ cannot be violated by the deletion of $o$. A similar argument by contradiction can be made for links and slots deleted by the slice. Therefore, Condition 2 is met. □

As ground concretizations are products, we have the following corollary:

**Corollary 1** (Universal conformance yields meta-model conformance)**.** Given a monotonic product line *MMPL* and a model $M$ s.t. $M \models_{type} MMPL$, $\forall MM_{\rho_i} \in$ *Prod(MMPL)* $\cdot M|_{MM_{\rho_i}} \models_{type_{\rho_i}} MM_{\rho_i}$.

Next, we show how to deal with non-monotonic MMPLs and propose concretization strategies that reduce the number of calls to a model finder.

### 5.2 Search Strategies for Partial Configurations

In this section, we extend the lifted analysis presented in Section 4 to deal with partial configurations. The overall process remains the same: we use a model finder to seek an instance of a FEMM, from which we extract a configuration $\rho$ and a model $M$ (see Fig. 7). However, we use a different FEMM than in Section 4 in order to obtain partial configurations as a result of the model search. This FEMM can be built using two strategies which we call *hard* and *soft*. Under the *hard* strategy, the model $M$ returned by the finder conforms to any ground concretization (i.e., a product meta-model) of the MMPL for the configuration. Under the *soft* strategy, we apply Def. 10 to produce slices of the model $M$ conformant to the different ground MMPL concretizations.

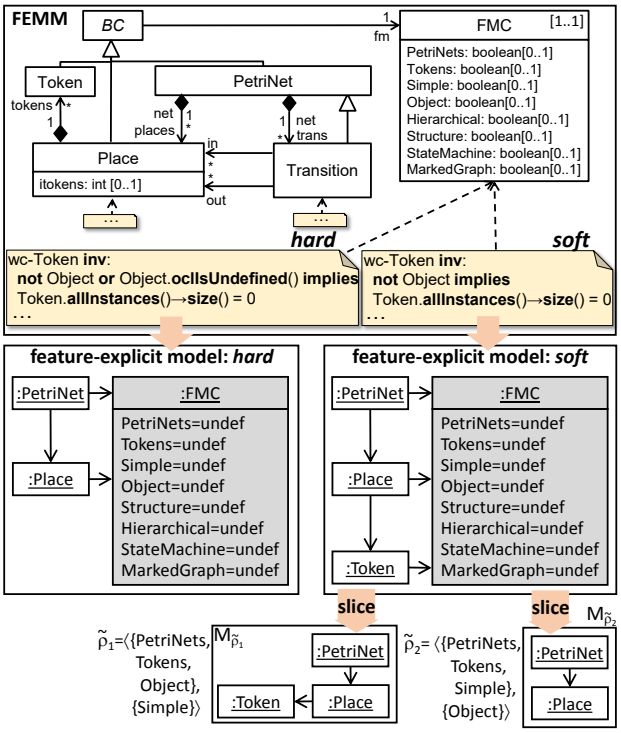



Fig. 13. Example of *hard* and *soft* search strategies.

**Example.** Fig. 13 illustrates the difference between the *hard* and *soft* strategies. The FEMM encodes the PCs and modifiers in the MMPL according to the strategy. The left side of the figure shows a model and a partial configuration returned by the finder, where all features are undefined. Since the strategy is *hard*, the model conforms to every meta-model of the MMPL. The model on the right side was obtained using the *soft* strategy and needs to be sliced for every ground concretization of the MMPL. The figure shows two such slices. $M_{\tilde{\rho}_2}$ corresponds to configurations where Object is *false*, and hence the Token object is deleted.

We use a modified version of the algorithm in Listing 1 to generate the FEMM (in the following, line numbers refer to this listing). Specifically, the attributes emulating features in class FMC are set cardinality [0..1], so that they can be *true*, *false* or *undefined* (line 10). Moreover, the invariants derived from PCs and modifiers are updated as follows to account for undefined features and the *hard* and *soft* strategies:

**PC of classes (line 17) and fields (lines 28–29).** The *hard* strategy requires that a class have no instances if its PC is *false* or *undefined* (i.e., (not pc or pc=undef) implies size(c)=0), and similarly for fields. The *soft* strategy requires empty fields and class instance sets only when their PCs are *false*, as in Listing 1. As an example, Fig. 13 shows the invariant wc-Token derived from the PC of Token for both strategies.
For non-monotonic MMPLs and the *soft* strategy, we require two additional conditions to prevent constructing non-conformant model slices. First, the operation size in the previous invariants needs to count the instances of the class but not of its subclasses. Second, if a class with a PC inherits a reference or it inherits from the target class of a reference, we use the *hard* strategy to build the invariant of the class. For example, in Fig. 12, class Module inherits from PetriNet which is the target of pn; hence, we use the *hard* strategy to derive the invariant (not Modular or Modular=undef) implies size(Module)=0. This way, models can only contain instances of Module when feature Modular is *true*, ensuring a correct slice.

**PC of invariants (lines 19, 43–44).** Both strategies require that an invariant holds if its PC is *true* (i.e., pc=true implies wc). However, when its PC is *undefined*, the *hard* strategy considers that the invariant does not hold (i.e., pc=undef implies false), while the *soft* strategy requires the invariant to hold (i.e., pc=undef implies wc). The former implies that no objects of a class with *undefined* PC get populated. The latter ensures universal conformance (cf. Def. 9).

**Cardinality modifiers (line 26).** We handle both strategies the same way by adding an extra case: if the condition in a cardinality modifier is *undefined*, then both the original feature cardinality (*f*.min) and the modifier cardinality (*m*) must be satisfied, as required by the notion of universal conformance in Def. 9. In the running example, it means that partial configurations where StateMachine is *undefined* require the size of references in and out to be at least 0 and equal to 1, the latter condition being stronger.

**Inheritance modifiers (lines 38, 40–41).** The *soft* strategy uses the same invariants as in Listing 1, while the *hard* strategy requires the consequences in the invariants to hold even when the premise is *undefined*.

**Formula in FMC (line 11).** Regardless of the strategy, we relax the formula implied by the feature model to avoid overconstraining the search and enable finding partial configurations. Hence, we only require the following: if a feature is *true*, then all of its mandatory children are *true*; if a feature is *undefined*, then all of its children are *undefined*; if an alternative-feature is *true*, then one of its children is *true* and the rest are *false*; and if an or-feature is *true*, then at least one of its children is *true*.

These changes to the algorithm in Listing 1 allow obtaining partial configurations as a result of the model finding. As explained in Section 4, finding all meta-models satisfying a property requires calling the model finder several times. When the finder returns a total configuration, we forbid it as a valid result in the next search. If the configuration is partial, then we also forbid any total configuration that can be built from the partial one.

Note that, in each search, the model finder can return any valid configuration. To reduce the number of model finder calls, we guide the search to start with the partial configurations which represent the biggest total configuration sets. With this goal, we use the method in Listing 2 to obtain the features that should be undefined or not in the next search, adding an invariant to class FMC to enforce it. The first search requires that the root feature in the feature model be *undefined* (line 4 in Listing 2). This way, if the search yields a solution, then the process ends as this partial configuration represents all total ones. Otherwise, we call the method again to obtain a less permissive partial configuration (lines 7–14). In such a case, the method receives all features set to *undefined* in the last method call (the first time is just the root feature), selects the one with the most children features (line 7), forces it not to be *undefined* (line 8), and sets all of its children features but the one with the least

```
1  nextPartialConfiguration (FM : in, undef : inout, def : inout)
2    // base case: set root feature to undefined
3    if size(undef)==0 and size(def)==0 then
4      add FM.rootFeature to undef
5    else if size(undef)>0 then
6      // set undefined feature with more children to defined
7      set f1 = feature with more descendants ∈ undef
8      move f1 from undef to def
9      // set its nonleaf child with fewer descendants to defined,
10     // and the rest of children to undefined
11     set f2 = nonleaf feature with fewer descendants ∈ f1.children
12     forall child ∈ f1.children do
13       if child == f2 then add f2 to def
14       else if size(child.children)>0 then add f2 to undef
```

Listing 2. Heuristic construction of partial configuration in each search.

children to *undefined* (lines 11-14). This heuristic maximizes the number of undefined features and obtains the remaining partial configurations that represent more total ones. Then, we modify the invariant in FMC reflecting those features that need to be undefined and perform a new search. This process is repeated as needed. Section 7.2 shows how this heuristic effectively reduces the number of invocations to the model finder in the lifted analysis.

**Example.** In our example, the first search looks for a configuration where the root feature is *undefined* (invariant PetriNets = undef). If a model is found (as is the case), then it is a witness to every product meta-model. Otherwise, the next search looks for a configuration where the root feature and exactly one of its children are not *undefined* (invariant not PetriNets = undef and not Tokens = undef and Structure = undef).

## 6 Tool Support

We have realized our approach as an Eclipse plugin called MERLIN, available at http://miso.es/tools/merlin. MERLIN extends FeatureIDE [35] to create feature models, handle configurations and build products. It integrates EMF for describing meta-models [27], the USE Validator [20] for model finding, and the Eclipse OCL project [36] and Sat4J [37] for the static analysis of OCL expressions [21].

Fig. 14 shows a screenshot of the tool. To build an MMPL, the user first needs to create a FeatureIDE project, selecting the MERLIN extension. Label 1 in the figure shows the package explorer with a few MERLIN projects. Then the user builds the feature model with FeatureIDE (label 2), and the *150MM* with the OCLinEcore meta-model editor (label 3). The PCs and modifiers are defined as annotations within the Ecore-based *150MM*s. The tool allows discovering unsatisfiable PCs and modifiers by relying on SAT solving and taking into account the formula implied by the feature model (see [21]). For example, a PC Simple ∧ Object is not satisfiable in the running example, which gets reported as an error in the Eclipse problem view (label 4).

Label 5 in the figure shows the wizard for the instance property analysis. Here the user can enter structural and mixed properties, and configure the different search and result options. Given a property, the tool can extract its implied PC, which can be used to reduce the search scope. The results returned by the model finder are parsed back into EMF models and FeatureIDE configuration files.

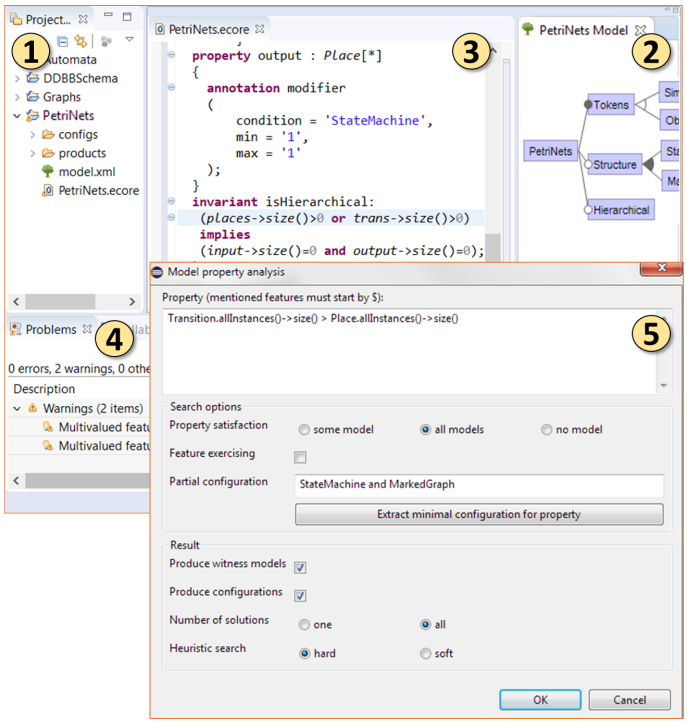



Fig. 14. Using MERLIN to analyse the running example.

## 7 Evaluation

This section reports on the evaluation of the scalability of our proposal, driven by the following research question: *Is the lifted property satisfiability analysis on the MMPL faster than analysing each meta-model in the MMPL separately?*

Looking at the analysis options in the feature model of Fig. 5, there is only one major factor that affects the analysis performance: whether the analysis must return just one meta-model of the MMPL satisfying the given property (Solutions=one) or all of them (Solutions=all). The rationale is that this makes a difference to the number of invocations to the solver that the analysis has to perform. Hence, the evaluation has two parts: Section 7.1 evaluates the lifted analysis when looking for one solution meta-model, and Section 7.2 focuses on the problem of looking for all meta-models. The latter experiment evaluates the core contribution of the paper, i.e., the efficiency gains of the lifted analysis with partial configurations, including our guided search strategies of partial configurations (cf. Section 5.2), compared to either using the lifted analysis with total configurations or analysing each meta-model in an MMPL separately. The MMPLs used in the evaluation and the raw data of the results are available at http://miso.es/tools/merlin/evaluation_tse.

### 7.1 Efficiency when Looking for One Meta-Model

Our first experiment deals with the problem of finding one meta-model having instances with a given property (i.e., Solutions=one in the analysis space of Fig. 5). As a representative case, we consider the property to analyse to be "true", and hence the analysis consists of finding a configuration that yields an instantiable meta-model. This way, we compare the performance of our lifted analysis to

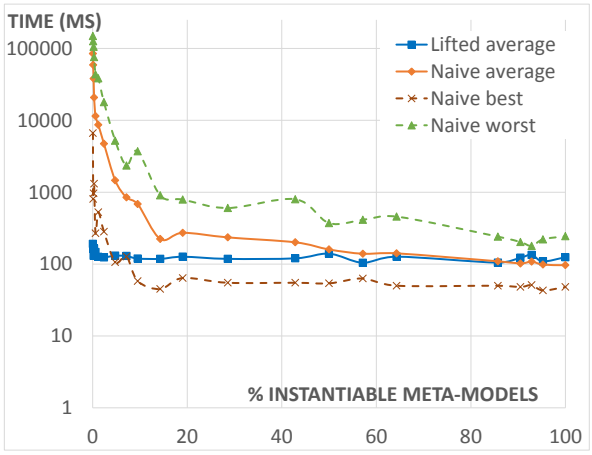



Fig. 15. Lifted and naive analyses, finding one solution meta-model.

find one instantiable meta-model in an MMPL, w.r.t. a *naive* enumerative approach that generates one meta-model of the MMPL at a time, checks its instantiability and finishes if it has instances. The sooner an instantiable meta-model is found, the faster the analysis concludes. Hence, the analysis time in the naive approach depends on the percentage of instantiable meta-models in the MMPL: if many meta-models are instantiable, the likelihood of finding one soon increases. For this reason, this experiment considers MMPLs with different instantiability ratios.

In particular, we use an MMPL created by us to represent automata, which comprises 2 016 meta-models. Its feature model has 20 features, and its *150MM* contains 6 classes, 5 invariants and 18 PCs. While all 2 016 meta-models in this MMPL have instances, we manually built 22 versions of the MMPL, each one producing a different percentage of instantiable meta-models (i.e., one version had one instantiable meta-model out of 2 016, another had two, and so on). This was done to emulate MMPLs with different meta-model instantiability ratios. Technically, we built each MMPL version by adding to the classes in its *150MM* the invariant "false", annotated with a PC satisfied by the desired percentage of possible feature configurations. Then we used both approaches (lifted and naive) to analyse MMPL instantiability of each version, 40 times each, computing the average time. We used the variant of the lifted analysis that does not use partial configurations, as it is sufficient to find one meta-model. In the naive approach, the traversal of the meta-models in the MMPL was randomized in each analysis. We conducted the experiment on a Windows 10 computer with i7-6500U processor and 16Gb of memory.

Fig. 15 shows the average analysis time of each approach in milliseconds (vertical axis in logarithmic scale) with respect to the ratio of instantiable meta-models in the MMPL (horizontal axis). It also shows the best and worst times for the naive approach, but not for the lifted one because the variance in this case is minimal.

We can see that the lifted analysis was faster than the naive one when fewer than 85% meta-models were instantiable, being 1 000x faster when this percentage was under 10%. In practice, this may occur when the language designer is creating an MMPL and defines an incorrect invariant, likely in the common part of the *150MM*, making many meta-model variants non-instantiable. Moreover, note that meta-model instantiability corresponds to analysing the satisfiability of the OCL property "true". However, we may analyse other OCL properties that few meta-models satisfy (e.g., disallow a negative number of tokens). These results show that our lifted analysis is most useful in such cases.

For MMPLs with more than 85% instantiable meta-models, the performance of the lifted analysis was slightly slower than the naive one but still reasonable (100ms vs 120ms). The best case for the naive approach corresponds to finding an instantiable meta-model at the first attempt, in which case, it was up to 3x faster than the lifted analysis because the constraint solving problem is easier. Hence, overall the lifted analysis was only slightly slower than the best case of the naive approach, but it was several orders of magnitude faster than the corresponding worst case.

*Threats to validity.* These results provide some evidence of the benefits of lifting the analysis of instance properties to the product line level. However, a threat to their generality is the low number of artefacts used in the evaluation, and the fact that we have created them synthetically by hand. Moreover, the experiment only considers the analysis of the property "true" (i.e., meta-model instantiability). However, we believe that analysing a different OCL-expressed property would yield a similar result, as this amounts to adding the property as another invariant to the FEMM, while the analysis method remains the same.

### 7.2 Efficiency when Looking for All Meta-Models

The goal of the second experiment is to assess whether using partial configurations in our lifted analysis entails an improvement in the analysis performance. For this purpose, we apply our lifted analysis and a *naive* enumerative approach to the problem of finding all instantiable meta-models in an MMPL (i.e., Solutions=all in Fig. 5). The naive approach implies generating and analysing the instantiability of every meta-model of the MMPL separately. The lifted approach implies building the FEMM and invoking the model finder iteratively, each time disallowing previously found configurations. We consider three variants for the lifted analysis: using total configurations, using partial configurations with the strategy *hard*, and using partial configurations with the strategy *soft*.

The eight MMPLs considered in the experiment are given in Table 2. The second to fourth columns show their complexity metrics: the number of features in the feature model (from 4 to 48); the number of product meta-models (from 5 to more than 2 million); and the number of classes, invariants, PCs and modifiers in the *150MM*. Three of these MMPLs were created by us (Automata, the running example and State machines); one was created from a set of existing meta-models (Relational DDBB); and four came from the literature (GPL, Graph algs, Process modelling and Role modelling). The Automata MMPL is the one used in the evaluation in Section 7.1. The Graph algs product line is a common benchmark in the SPL community [38], [41], but as we focus on the language and not on the algorithms, we included features related to its structure and some invariants. We created the Relational DDBB MMPL from nine third-party meta-models

TABLE 2
Lifted and naive analyses, finding all solution meta-models.

| Name | Feats | MMs | 150MM classes/invs/ PCs/modifs | Analysis time | | | | Invocations to solver | | | | Different instances | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Naive | Lifted total | Lifted partial hard | Lifted partial soft | Naive | Lifted total | Lifted partial hard | Lifted partial soft | Naive | Lifted total | Lifted partial hard | Lifted partial soft |
| Automata | 20 | 2 016 | 6/5/18/0 | 174.7s | >5h | **2.7s** | 3.1s | 2 016 | 2 017 | 1 | 1 | 824 | 1 378 [a] | 1 | 1 |
| GPL [38] | 38 | 840 | 16/6/42/4 | 108.4s | 1 656.9s | 5.7s | **3.2s** | 840 | 841 | 15 | 1 | 524 | 840 | 2 | 12 |
| Graph algs [39] | 15 | 192 | 4/1/8/3 | 19s | 55.1s | 2.4s | **2.3s** | 192 | 193 | 1 | 1 | 29 | 192 | 1 | 2 |
| Process modelling [40] | 18 | 960 | 11/2/9/1 | 140.3s | 4 908.3s | **2.3s** | 2.4s | 960 | 961 | 1 | 1 | 260 | 960 | 1 | 1 |
| Relational DDBB | 10 | 24 | 7/0/17/0 | 4.1s | 4.6s | **2.2s** | 2.3s | 24 | 25 | 1 | 1 | 24 | 24 | 1 | 24 |
| Role modelling [16] | 48 | >2 395 000 | 40/0/32/9 | >5h | 3 807s [e] | 4.9s | **4.1s** | >2 395 000 | 1 491 [e] | 1 | 1 | 13 [b] | 1 490 [c] | 1 | 4 [d] |
| Running example | 8 | 16 | 4/2/5/3 | 4.9s | 3.7s | 2.5s | **2.3s** | 16 | 17 | 1 | 1 | 16 | 16 | 1 | 1 |
| State machines | 4 | 5 | 5/0/5/0 | 3s | 3.9s | **2.2s** | 2.5s | 5 | 6 | 1 | 1 | 5 | 5 | 1 | 2 |

[a] Calculated over the set of 1 378 models generated in 5 hours.
[b] Calculated over the set of 3 405 models generated in 5 hours.
[c] Calculated over the set of 1 490 models generated in 3 807 seconds.
[d] Calculated over the set of 4 178 models generated in 5 hours.
[e] Data collected until the constraint solver crashes with a memory exception.

available in the ATL meta-model zoo[3]. Altogether, the evaluation includes MMPLs of different sizes (small, medium and large) and provenance (constructed by us and from the literature).

In the experiment, the search space of the model finder was configured with a bound of at most five instances of each class, and a timeout of five hours. To discard cache effects, we restarted the system after each analysis.

Table 2 shows the results obtained for each approach: analysis time, the number of invocations to the solver (i.e., the number of model finding problems solved), and the number of different instance models produced by the analysis.

The lifted analyses with partial configurations (Lifted partial hard and Lifted partial soft) were always the fastest, with an analysis time generally lower than 5 seconds. The speedup w.r.t. the naive approach ranged between 1.3x (State machines) to 63x (Automata), being more than 4 300x for Role modelling. Actually, the naive analysis of the latter MMPL did not finish within the five hours timeout, while it took less than 5 seconds using the lifted analysis with partial configurations. Both strategies *hard* and *soft* had similar performances except for GPL. The reason is that the *hard* strategy required invoking the solver 15 times whereas the *soft* strategy did it just once. In general, the *hard* strategy requires more invocations to the solver than *soft* whenever the meta-models in the MMPL do not share instances.

The lifted analysis based on total configurations had by far the worst performance and did not finish in two cases: Automata due to the timeout, and Role modelling due to a memory exception raised by the model finder. The lifted analysis using partial configurations was up to 2 000 times faster than using only total configurations.

Finally, we look at the diversity of the produced witness models. The naive approach makes one model finder call per each meta-model in the MMPL; however, on average, only one third of the models returned by the finder were different. Strategies *hard* and *soft* invoked the finder just once (except for GPL); however, while the *hard* strategy produced only a (very simple) witness model, the *soft* one resulted in a variety of more intricate and diverse models. Lifted analysis with total configurations produced the most diverse model set.

3. http://web.emn.fr/x-info/atlanmod/index.php?title=Ecore

*Threats to validity.* The experiment shows that lifting the instantiability analysis to the MMPL and considering partial configurations improves the analysis performance for the eight MMPLs in our setup. Although the evaluation uses a low number of MMPLs constructed by us, they have widely varying sizes and origin, with half of them coming from the literature. This reduces the risk that our findings cannot be generalized. As in the previous experiment, we have evaluated meta-model instantiability but not the other property types; however, we believe that the results would not be significantly different.

## 8 Related Work

Next, we review the three main lines of related work: model-based PLs, variability in modelling languages, and analysis techniques for model-based PLs.

*Model-based PLs.* SPLs allow expressing variability in MDE artefacts. For example, model-based product lines can be an intermediate step towards code generation [42], and some companies (e.g., in defence, embedded and automotive domains) use SPLs to manage the variability across their model assets [8], [9], [10]. Other researchers have proposed PLs for models of specific formalisms, like uses cases [43], Petri nets [44] or Statecharts [45]. All these works apply SPLs to specific modelling languages, at the model level. Instead, we work at the meta-model level, defining PLs of languages.

*Variability in modelling languages.* Model-driven solutions based on the use of DSLs and code generators can be seen as an open PL mechanism to automate software production [46]. While most of these solutions have a fixed DSL, many researchers have recognized the need to express variability in modelling languages [11], [41], [47], [48], [49]. PLs have been defined for languages like Petri nets [21], architectural languages [49], state machines [50], feature modelling notations [51] and role-based modelling languages [16], among many others. The UML specification includes informal but explicit *semantic variation points* to address different usage contexts [50], [52]. To tackle this issue, [53] proposed meta-model templates to enable language specialization and composition via parameter binding.

With the aim of offering a closed set of language variants, some language engineering approaches propose *documenting* semantic variations as feature models [54]. Beyond mere

documentation, MMPLs have been applied to the reuse of domain-specific modelling languages [4] and their transformations [17]. Language workbenches for the definition of textual languages, like MontiCore [47], [55] support the definition of language PLs. Unfortunately, these works lack means to ensure that the resulting language definitions are correct. However, building a language PL can be challenging and error-prone if there are many features or invariants to consider. This makes analysis techniques like the ones proposed in this paper necessary.

A key ingredient of model-based SPLs is the way to express model variability and automate product derivation. Some UML-specific approaches [43] rely on stereotypes to express variability [56] or modify the language definition in an intrusive way [44]. In contrast, approaches such as delta-modelling [15], [16], [57] and VML* [25] favour the separation of concerns. They are language-independent and can be applied to SPLs of models or meta-models. Moreover, within the ABS language, delta-modelling has also been applied to express variability of code bases [57]. The focus of those approaches is on expressivity, which enables the use of arbitrary operations to synthesize a product, though this makes ensuring syntactic and semantic correctness challenging. Instead, our approach based on PCs and modifiers facilitates analysis and enables instance property analysis by model finding, at the cost of expressiveness. However, without analysis techniques, feature incompatibilities (e.g., StateMachine and Hierarchical in our running example) may go unnoticed and percolate to the final system, creating errors.

*Analysis of model-based SPLs.* Many analysis techniques for SPLs have been proposed. Von Rehin et al. [58] argue that the analysis of each variant in an SPL cannot be made efficiently with standard techniques. This has given rise to *variability-aware* techniques that lift analyses from single artefacts to the product line level, and *sampling* heuristics to analyse a subset of all derivable products, like prioritization [59], [60] or pairwise testing (see [61] for a classification and survey). Both syntactic [62], [63] and behavioural analyses (e.g., model checking [64], [65], [66] and domain-specific analyses such as for component fault diagrams [67]) have been lifted to SPLs. For programming languages, static analyses based on the control-flow graph have been lifted to SPLs [68], [69]. As demonstrated in [58], the advantages of variability-aware techniques such as lifting are a better efficiency and efficacy than enumerative approaches. Moreover, they enable richer specification means like mixed properties.

Some works analyse well-formedness of PLs of models, at the PL level. For example, [45] uses delta-modelling to define PLs of statecharts and analyses well-formedness of each derivable statechart at the PL level. In a more general setting, PLs of models are analysed to ensure that each model conforms to its meta-model and fulfils its integrity constraints in [18]. This corresponds to a syntactic analysis lifting. Instead, our lifted analyses tackle meta-models and ensure that each product meta-model has the required instantiability properties. Moreover, we characterize the instantiability analyses that can be done at the MMPL level.

In [70], the authors present the tool Clafer, which unifies meta-models and feature models. It supports the lifted analysis of the instantiability of the meta-models in an MMPL via a compilation into Alloy. However, it does not support lifted analyses of other instance properties, as we do in our paper. In [71], the authors synthesize random, erroneous model-based SPLs for a modelling language, to showcase typical errors in the SPL design.

Altogether, we note that analysis techniques for MMPLs at the PL level are currently lacking. For this reason, we have lifted meta-model validation techniques based on constraint solving to MMPLs. To the best of our knowledge, the lifting process using both total and partial configurations and the classification of instance property analyses are novel contributions of our work.

## 9 Conclusions

In this paper, we have argued for the need to analyse instantiability properties of language product lines. For this purpose, we have proposed a classification of instance property types, and then lifted the property analysis to MMPLs, optimized by the use of partial configurations. We have implemented the approach in a tool called MERLIN, and reported on experiments showing the scalability benefits w.r.t. an explicit analysis of each product in the MMPL.

This work opens the door to a wider use of variability techniques within MDE to foster reusability. For instance, building on MMPLs would allow the creation of product lines of model transformations [39], code generators and model editors, reusable for each meta-model of the MMPL.

Our approach currently supports modifiers for field cardinalities and inheritance, but to improve expressiveness, we plan to include other modifiers to control the abstractness of classes or the type of fields. We also plan to expand our analyses to check subsumption of language variants. For example, in the case of Petri nets, all StateMachine nets are also FreeChoice nets, but not vice versa [1]. This analysis will help in constraining the feature model to reduce the configuration space and reflect expected language relations. Finally, performing a user study is also future work.

## Acknowledgments

This work has been funded by the Spanish Ministry of Science (RTI2018-095255-B-I00), the R&D programme of Madrid (P2018/TCS-4314), and by NSERC. We thank the anonymous referees for their useful comments.

## References

[1] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.
[2] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, Feb. 2006.
[3] OCL, http://www.omg.org/spec/OCL/, 2014.
[4] J. White, J. H. Hill, J. Gray, S. Tambe, A. S. Gokhale, and D. C. Schmidt, "Improving domain-specific language reuse with software product line techniques," *IEEE Software*, vol. 26, no. 4, pp. 47–53, 2009.
[5] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *IEEE Trans. Software Eng.*, vol. 39, no. 6, pp. 869–891, 2013.
[6] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering. Foundations, Principles and Techniques*. Springer-Verlag Berlin Heidelberg, 2005.

[7] F. J. v. d. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag, 2007.

[8] B. Young, J. Cheatwood, T. Peterson, R. Flores, and P. C. Clements, "Product line engineering meets model based engineering in the defense and automotive industries," in *Proc. of SPLC'17*. ACM, 2017, pp. 175–179.

[9] A. Schlie, D. Wille, S. Schulze, L. Cleophas, and I. Schaefer, "Detecting variability in MATLAB/Simulink models: An industry-inspired technique and its evaluation," in *Proc. of SPLC'17*. ACM, 2017, pp. 215–224.

[10] S. Trujillo, J. M. Garate, R. E. Lopez-Herrejon, X. Mendialdua, A. Rosado, A. Egyed, C. W. Krueger, and J. D. Sosa, "Coping with variability in model-based systems engineering: An experience in green energy," in *Proc. of ECMFA'10*, ser. LNCS, vol. 6138. Springer, 2010, pp. 293–304.

[11] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, and B. Baudry, "Leveraging software product lines engineering in the development of external DSLs: A systematic literature review," *Comp. Langs., Systems & Structures*, vol. 46, pp. 206–235, 2016.

[12] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-90-TR-021, 1990.

[13] J. Cabot, R. Clarisó, and D. Riera, "On the verification of UML/OCL class diagrams using constraint programming," *Journal of Systems and Software*, vol. 93, pp. 1–23, 2014.

[14] F. Basciani, J. di Rocco, D. di Ruscio, L. Iovino, and A. Pierantonio, "Automated clustering of metamodel repositories," in *Proc. of CAiSE'16*, ser. LNCS, vol. 9694. Springer, 2016, pp. 342–358.

[15] C. Seidl, I. Schaefer, and U. Aßmann, "DeltaEcore – a model-based delta language generation framework," in *Modellierung*, ser. LNI, vol. 225. Bonn: GI, 2014, pp. 81–96.

[16] T. Kühn, M. Leuthäuser, S. Götz, C. Seidl, and U. Aßmann, "A metamodel family for role-based modeling and programming languages," in *Proc. of SLE'14*, ser. LNCS, vol. 8706. Springer, 2014, pp. 141–160.

[17] G. Perrouin, M. Amrani, M. Acher, B. Combemale, A. Legay, and P. Schobbens, "Featured model types: Towards systematic reuse in modelling language engineering," in *Proc. of MiSE@ICSE'16*. New York, NY, USA: ACM, 2016, pp. 1–7.

[18] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness OCL constraints," in *Proc. of GPCE'06*. New York, NY, USA: ACM, 2006, pp. 211–220.

[19] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. London, England: MIT Press, 2006, see also http://alloy.mit.edu/.

[20] M. Kuhlmann and M. Gogolla, "From UML and OCL to relational logic and back," in *Proc. of MODELS'12*, ser. LNCS, vol. 7590. Springer, 2012, pp. 415–431.

[21] E. Guerra, J. de Lara, M. Chechik, and R. Salay, "Analysing metamodel product lines," in *Proc. of SLE'18*. ACM, 2018, pp. 160–173.

[22] R. Salay, M. Famelis, J. Rubin, A. D. Sandro, and M. Chechik, "Lifting model transformations to product lines," in *Proc. of ICSE'14*. New York, NY, USA: ACM, 2014, pp. 117–128.

[23] S. Apel, F. Janda, S. Trujillo, and C. Kästner, "Model superimposition in software product lines," in *Proc. of ICMT'09*, ser. LNCS, vol. 5563. Springer, 2009, pp. 4–19.

[24] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *Proc. of GPCE'05*, ser. LNCS, vol. 3676. Springer, 2005, pp. 422–437.

[25] S. Zschaler, P. Sánchez, J. P. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. Araújo, and U. Kulesza, "VML* - a family of languages for variability management in software product lines," in *Proc. of SLE'09*, ser. LNCS, vol. 5969. Springer, 2009, pp. 82–102.

[26] MOF, http://www.omg.org/spec/MOF, 2016.

[27] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework, $2^{nd}$ Edition*. Upper Saddle River, NJ: Addison-Wesley Professional, 2008.

[28] OCLinEcore, https://wiki.eclipse.org/OCL/OCLinEcore, 2018.

[29] Xcore, https://wiki.eclipse.org/Xcore, 2018.

[30] EMFatic, https://www.eclipse.org/emfatic/, 2012.

[31] C. A. González, F. Büttner, R. Clarisó, and J. Cabot, "EMFtoCSP: A tool for the lightweight verification of EMF models," in *Proc. of FormSERA'12*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 44–50.

[32] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using boolean satisfiability," in *Proc. of DATE'10*. IEEE Computer Society, 2010, pp. 1341–1344.

[33] M. Balaban and A. Maraee, "Finite satisfiability of UML class diagrams with constrained class hierarchy," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 24:1–24:42, 2013.

[34] S. MacLane, *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[35] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake, *Mastering Software Variability with FeatureIDE*. Springer, 2017, see also https://featureide.github.io/.

[36] Eclipse OCL project, http://wiki.eclipse.org/OCL, 2018.

[37] D. L. Berre and A. Parrain, "The Sat4j library, release 2.2," *JSAT*, vol. 7, no. 2-3, pp. 59–6, 2010.

[38] R. E. Lopez-Herrejon and D. Batory, "A standard problem for evaluating product-line methodologies," in *Proc. of GCSE'01*, ser. LNCS, vol. 2186. Springer, 2001, pp. 10–24.

[39] J. de Lara, E. Guerra, M. Chechik, and R. Salay, "Model transformation product lines," in *Proc. of MODELS'18*. ACM, 2018, pp. 67–77.

[40] J. S. Cuadrado, E. Guerra, and J. de Lara, "A component model for model transformations," *IEEE Trans. Software Eng.*, vol. 40, no. 11, pp. 1042–1060, 2014.

[41] T. Buchmann and F. Schwägerl, "Advancing negative variability in model-driven software product line engineering," in *ENASE Revised Selected Papers*, ser. Comm. in Comp. and Inf. Sci., vol. 703. Springer, 2016, pp. 1–26.

[42] S. Trujillo, D. S. Batory, and O. Díaz, "Feature oriented model driven development: A case study for portlets," in *Proc. of ICSE'07*. IEEE Computer Society, 2007, pp. 44–53.

[43] I. Hajri, A. Goknil, L. C. Briand, and T. Stephany, "Configuring use case models in product families," *SoSyM*, vol. 17, no. 3, pp. 939–971, 2018.

[44] R. Muschevici, J. Proença, and D. Clarke, "Feature nets: behavioural modelling of software product lines," *SoSyM*, vol. 15, no. 4, pp. 1181–1206, 2016.

[45] M. Lienhardt, F. Damiani, L. Testa, and G. Turin, "On checking delta-oriented product lines of statecharts," *Sci. Comput. Program.*, vol. 166, pp. 3–34, 2018.

[46] J. Tolvanen and S. Kelly, "Defining domain-specific modeling languages to automate product derivation: Collected experiences," in *Proc. of SPLC'05*, ser. LNCS, vol. 3714. Springer, 2005, pp. 198–209.

[47] M. V. Cengarle, H. Grönniger, and B. Rumpe, "Variability within modeling language definitions," in *Proc. of MODELS'09*, ser. LNCS, vol. 5795. Springer, 2009, pp. 670–684.

[48] N. I. Altintas, S. Cetin, A. H. Dogru, and H. Oguztüzün, "Modeling product line software assets using domain-specific kits," *IEEE Trans. Software Eng.*, vol. 38, no. 6, pp. 1376–1402, 2012.

[49] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann, "Modeling language variability with reusable language components," in *Proc. of SPLC'18*. ACM, 2018, pp. 65–75.

[50] A. Taleghani and J. M. Atlee, "Semantic variations among UML statemachines," in *Proc. of MODELS'06*, ser. LNCS, vol. 4199. Springer, 2006, pp. 245–259.

[51] C. Seidl, T. Winkelmann, and I. Schaefer, "A software product line of feature modeling notations and cross-tree constraint languages," in *Proc. of Modellierung*, ser. LNI, vol. P-254. GI, 2016, pp. 157–172.

[52] F. Chauvel and J. Jézéquel, "Code generation from UML models with semantic variation points," in *Proc. of MODELS'05*, ser. LNCS, vol. 3713. Springer, 2005, pp. 54–68.

[53] A. Cuccuru, C. Mraidha, F. Terrier, and S. Gérard, "Templatable metamodels for semantic variation points," in *Proc. of ECMDA-FA'07*, ser. LNCS, vol. 4530. Springer, 2007, pp. 68–82.

[54] H. Grönniger and B. Rumpe, "Modeling language variability," in *Proc. of Monterey Workshop on Foundations of Computer Software*, ser. LNCS, vol. 6662. Springer, 2010, pp. 17–32.

[55] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann, "Systematic composition of independent language features," *Journal of Systems and Software*, vol. 152, pp. 50–69, 2019.

[56] T. Ziadi and J. Jézéquel, "Software product line engineering with the UML: deriving products," in *Software Product Lines - Research Issues in Engineering and Management*. Springer, 2006, pp. 557–588.

[57] D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte, "Variability modelling in the ABS language," in *Proc. of FMCO'10*, ser. LNCS, vol. 6957. Springer, 2010, pp. 204–224.

[58] A. von Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, "Variability-aware static analysis at scale: An empirical study," *ACM Trans. Sof. Eng. Meth.*, vol. 27, no. 4, pp. 18:1–18:33, 2018.
[59] X. Devroey, G. Perrouin, M. Cordy, H. Samih, A. Legay, P. Schobbens, and P. Heymans, "Statistical prioritization for software product line testing: an experience report," *SoSyM*, vol. 16, no. 1, pp. 153–171, 2017.
[60] M. Al-Hajjaji, T. Thüm, M. Lochau, J. Meinicke, and G. Saake, "Effective product-line testing using similarity-based product prioritization," *SoSyM*, vol. 18, no. 1, pp. 499–521, 2019.
[61] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 6:1–6:45, 2014.
[62] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. S. Batory, "Guaranteeing syntactic correctness for all product line variants: A language-independent approach," in *Proc. of TOOLS EUROPE'09*, ser. LNBIP, vol. 33. Springer, 2009, pp. 175–194.
[63] C. Kästner, S. Apel, T. Thüm, and G. Saake, "Type checking annotation-based product lines," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 3, pp. 14:1–14:39, 2012.
[64] M. H. ter Beek, E. P. de Vink, and T. A. C. Willemse, "Family-based model checking with mCRL2," in *Proc. of FASE'17*, ser. LNCS, vol. 10202. Springer, 2017, pp. 387–405.
[65] A. S. Dimovski and A. Wasowski, "Variability-specific abstraction refinement for family-based model checking," in *Proc. of FASE'17*, ser. LNCS, vol. 10202. Springer, 2017, pp. 406–423.
[66] M. Lochau, J. Bürdek, S. Hölzle, and A. Schürr, "Specification and automated validation of staged reconfiguration processes for dynamic software product lines," *SoSyM*, vol. 16, no. 1, pp. 125–152, 2017.
[67] C. Seidl, I. Schaefer, and U. Aßmann, "Variability-aware safety analysis using delta component fault diagrams," in *Proc. of SPLC workshops*. ACM, 2013, pp. 2–9.
[68] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini, "Spl$^{\text{lift}}$: statically analyzing software product lines in minutes instead of years," in *Proc. of PLDI'13*. ACM, 2013, pp. 355–364.
[69] F. Angerer, P. Grünbacher, H. Prähofer, and L. Linsbauer, "An experiment comparing lifted and delayed variability-aware program analysis," in *Proc. of ICSME'17*. IEEE Computer Society, 2017, pp. 148–158.
[70] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski, "Clafer: Unifying class and feature modeling," *SoSyM*, vol. 15, no. 3, pp. 811–845, 2016.
[71] J. B. F. Filho, O. Barais, M. Acher, J. Le Noir, A. Legay, and B. Baudry, "Generating counterexamples of model-based software product lines," *STTT*, vol. 17, no. 5, pp. 585–600, 2015.

**Esther Guerra** is Professor at the Universidad Autónoma in Madrid, where she leads the "Modelling and Software Engineering" research group (http://www.miso.es) together with J. de Lara. She has been a doctoral researcher at TU Berlin and "Sapienza" University of Rome, and a visiting professor at the Universities of York, Toronto and McGill. She is interested in Model-Driven Engineering, primarily in model transformations, transformation testing, meta-modelling and domain-specific languages.

**Juan de Lara** is Professor at the Universidad Autónoma of Madrid, where he works in Model-driven Engineering. He has published more than 200 papers in international journals and conferences and has been the PC co-Chair of ICMT12, FASE12, ICGT17, and will be the PC co-Chair of MODELS20 and SLE20. He is associate editor of the Journal on Software and Systems Modeling, JOT and IET Software.

**Marsha Chechik** is Professor at the University of Toronto. She is interested in the application of formal methods to improve the quality of software. She is a member of IFIP WG 2.9 on Requirements Engineering and an associate editor in chief of Journal on Software and Systems Modeling. She has been an associate editor of IEEE TSE in 2003-07 and 2010-13. She has been PC co-Chair of ICSE18, TACAS16, VSTTE16, ASE14, CONCUR08, CASCON08 and FASE09. She will be PC co-Chair of ESEC/FSE21.

**Rick Salay** is a researcher in software modeling with over 40 peer-reviewed papers in the area. He has conducted and led internationally recognized research in modelling on topics including safety assurance modeling, model management, model uncertainty and model transformations. Currently he plays senior roles in projects related to the safety of automated driving systems and machine learning.