Analysing Meta-Model Product Lines

Esther Guerra
Juan de Lara
Universidad Autónoma de Madrid
Spain

Marsha Chechik
Rick Salay
University of Toronto
Canada

Abstract

Model-driven engineering advocates the use of models to describe and automate many software development tasks. The syntax of modelling languages is defined by meta-models, making them essential artefacts. A combination of product line engineering methods and meta-models has been proposed to enable specification of modelling language variants, e.g., to describe a range of systems. However, there is a lack of techniques for ensuring syntactic correctness of all meta-models within a family (including their OCL constraints), and semantic correctness related to properties of individual instances of the different variants. The absence of verification methods at the product-line level can cause synthesis of ill-formed meta-models and problematic feature combinations whose effect at the instance level may go unnoticed.

To attack this problem, we propose an approach to lifting both the meta-model syntax checking and the satisfiability checking of properties of individual meta-model instances, to the product-line level. We validate the approach via a prototype tool called Merlin, and report on several experiments that show the advantages of our method w.r.t. an enumerative analysis approach.

CCS Concepts • **Software and its engineering → Software product lines**; *Software design engineering*; *Formal software verification*;

Keywords Model-Driven Engineering, Product Lines, Meta-Modelling, OCL, Model Finding

ACM Reference Format:
Esther Guerra, Juan de Lara, Marsha Chechik, and Rick Salay. 2018. Analysing Meta-Model Product Lines. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3276604.3276609

SLE '18, November 5–6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6029-6/18/11…$15.00
https://doi.org/10.1145/3276604.3276609

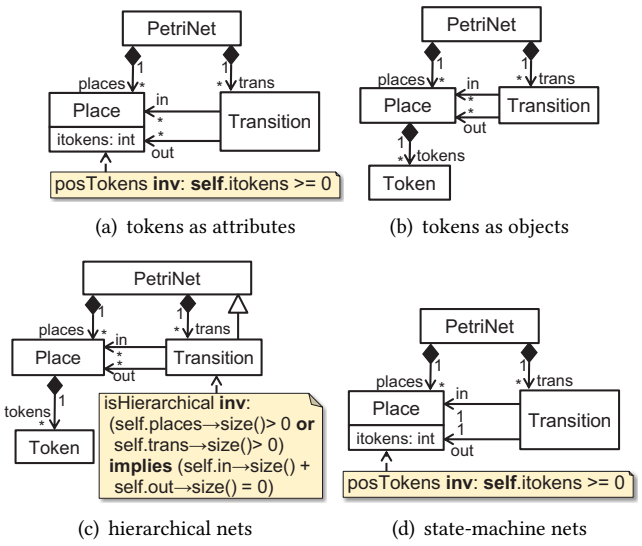
1 Introduction

Model-driven engineering (MDE) raises the abstraction level at which software is built, by promoting models as first-class artefacts during development [5, 40]. Models are therefore used to specify, verify, validate and generate code for the final system, among other activities. Models are created using the most appropriate language, either general-purpose (e.g., UML or Petri nets) or domain-specific (DSLs), explicitly tailored to define a range of systems within a domain. In MDE, the abstract syntax of a modelling language is described by a meta-model which defines domain primitives, their characteristics, relations and constraints. The latter are often expressed using the Object Constraint Language (OCL) [34].

Modelling languages may admit variants, so that they fit better for a project or scenario [49]. For example, we may offer variants of class diagrams for different purposes: while class operations or interfaces are not needed for analysis, we may include features 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 search [37].

Software product lines (SPLs) [36] have been proposed to express and manage collections of related software systems and their variability. As a particular case, they have been applied to language variability [31, 49] at the level of concrete syntax (i.e., visual representation), abstract syntax and semantics [31]. We focus on abstract syntax as meta-models are the core artefacts over which concrete syntax and semantics are defined. SPLs enable a compact representation of the language variants, providing an interface – *a feature model* [23] – for configuring a concrete language variant.

As an example, suppose our aim is to build a meta-model product line (MMPL) to describe variants of Petri nets [33]. Fig. 1 shows the meta-model of some variants. Variants in Figs. 1(a) and 1(b) account for two alternative realizations of tokens (as attributes or objects, respectively); the variant in Fig. 1(c) describes hierarchical nets (where hierarchical transitions contain both places and transitions); and the one in Fig. 1(d) describes state-machine nets (where transitions have exactly one input and one output). The targeted MMPL would encompass all these meta-model variants together with a feature model governing the presence/absence of meta-model elements depending on the selected features.



(a) tokens as attributes (b) tokens as objects

(c) hierarchical nets (d) state-machine nets

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

Ensuring the correctness of all meta-models in an MMPL is crucial, but analysing each meta-model separately may be expensive if there are many of them. Hence, this paper aims to answer two questions: (Q1) How to effectively create an MMPL ensuring that the meta-models and their OCL constraints across each allowed variant are syntactically correct? and (Q2) How to efficiently ensure that all instance models of meta-models within the MMPL have desired properties, such as meaningful combinations of objects, while being safe with respect to undesirable properties or configurations?

In the Petri nets example, answering Q1 means guaranteeing that in every meta-model variant that contains the constraint isHierarchical, transitions are subtypes of PetriNet, as otherwise, the access to references places and trans yields an error. Answering Q2 would help identify admissible meta-model combinations, e.g., whether we can combine hierarchical and state-machine nets. These two variants conflict if their integrity constraints clash, preventing creating any model, or precluding the use of the primitives each variant offers (e.g., if we are unable to create hierarchical transitions).

Although some approaches to MMPLs exist [27, 35, 41, 49], we are not aware of analysis techniques to ensure syntactic correctness of all meta-model variants, meaningful combination of variants, and effective specification and analysis of properties pertaining to a subset of the languages of the family. While some approaches analyse product lines of models [11], e.g., to check whether each product is well-formed (i.e., conforming to its meta-model), our focus is on product lines of *meta-models*. In that context, our goal is broader: not only to guarantee that the meta-models are well-formed, but also that they satisfy various *instantiability* properties (e.g., that the set of instance models is non-empty).

The contributions of this paper are the following: (1) A novel declarative notion of MMPL which supports OCL well-formedness constraints and is more amenable to automated analysis than the existing approaches [27, 41]. (2) Techniques for lifting syntactic checks of meta-models and their OCL constraints to the product-line level. These techniques use SAT solving and answer Q1. (3) Techniques for lifting satisfiability checking of (un)desired properties of meta-model instances to the product-line level, answering Q2. The lifting relies on an embedding of the feature model within a meta-model, and the use of model finders [21, 26] to analyse the satisfaction of properties expressed in OCL. We also provide a classification of MMPL property types including both lifted properties and *mixed* properties. The latter can refer to features of the feature model and be analysed for several meta-models of the MMPL (called *family-based specifications* [46]). (4) An evaluation of the effectiveness of our lifted analyses by comparing their performance w.r.t. an explicit analysis of each meta-model of the MMPL. (5) A prototype tool, called Merlin, available at http://miso.es/tools/merlin.

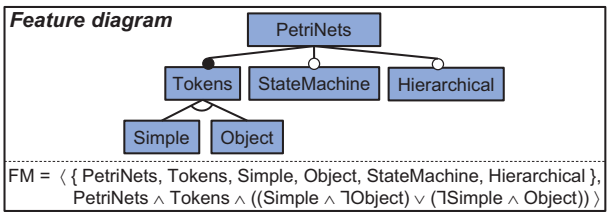


Figure 2. Feature model for the Petri nets example.

Paper organization. Sec. 2 introduces MMPLs. Sec. 3 describes lifting of syntactic constraints, allowing us to answer Q1. Sec. 4 describes lifting of satisfiability of instance model properties, allowing us to answer Q2. Sec. 5 presents tooling, and Sec. 6 evaluates its effectiveness. Sec. 7 discusses related work, and Sec. 8 presents conclusions and future work.

2 Meta-Model Product Lines

This section introduces the notions of MMPL, configuration and product derivation (in our context, a meta-model derivation), using the running example as illustration. We adopt an annotative approach which relies on known definitions of feature models [38]. While some approaches encode variability using full-fledged programming languages [27, 41, 52], we opt for annotations because they facilitate lifted analyses of MMPLs (see Secs. 3 and 4).

Def. 1 (Feature model [38]) A *feature model* $FM = (F, \phi)$ consists of a set of features $F = \{f_1, ..., f_n\}$ and a propositional formula ϕ that defines the allowable feature configurations.

Example. Fig. 2 shows a feature model representing the types of Petri nets in our running example (cf. Fig. 1). The upper part depicts the feature model using the classical feature diagram notation [23], and the bottom gives its representation using Def. 1. The feature model requires choosing a representation for tokens (Simple or Object, which are alternative), while it is possible to optionally select any combination of StateMachine and Hierarchical.

We base MMPLs in a standard notion of meta-model [32]:

Def. 2 (Meta-model) A *meta-model* is a tuple $MM = (C, FI, I, WC)$, 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 is owned by exactly one class and has cardinality interval $[min..max]$, with ∞ used for unbounded upper limits (i.e., $max = *$).
- $I \subseteq C \times C$ is the class inheritance relation.
- WC is a set of well-formedness constraints called *invariants*, each of which is assigned to exactly one class.

Non-abstract classes define the types of elements that can be 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 that point 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. In practice, we define invariants using OCL [34]. These are defined in the context of a class, and evaluated on all its instances.

Example. Fig. 1 shows four meta-models. The meta-model in Fig. 1(a) has three non-abstract classes, and four references with cardinality $[0..*]$. Class Place has an attribute itokens of type int, and an OCL invariant called posToken requiring every Place to have a non-negative value for itokens. Class Transition in Fig. 1(c) also has an invariant, isHierarchical, demanding every Transition with non-empty places or trans collections to have empty collections of input and output places (references in and out).

A necessary condition for a meta-model to have instances (i.e., models) is for it to be well-formed.

Def. 3 (Meta-model well-formedness) A meta-model has *well-formed structure* iff (1) every field is owned by a class, (2) every reference points to a class, (3) cardinality and inheritance are uniquely determined, and (4) there are no inheritance cycles. A meta-model is *well-formed* iff it has a well-formed structure, and the invariants are syntactically correct.

Next, we build on Defs. 1, 2 and 3 to propose the notion of *a meta-model product line* which would allow us to represent the meta-model variants of a family of languages.

Def. 4 (Meta-model product line) A *meta-model product line* is a tuple $MMPL = (FM, MM, \Phi, M_C, M_I)$, where[1]:

- $FM = (F, \phi_{MMPL})$ is a feature model.
- $MM = (C, FI, I, WC)$ is a meta-model with well-formed structure, called the 150% meta-model (*150MM* in short) [2, 10].
- Φ is a tuple of mappings $(\Phi_C, \Phi_{FI}, \Phi_{WC})$ from the feature model to the *150MM*. Each mapping Φ_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 Φ_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., we have an implication $\Phi_f \Rightarrow \Phi_{C_i}$), and same for invariants.
- M_C is a tuple of sets of *cardinality modifiers* (μ_{min}, μ_{max}). The set μ_{min} (resp. μ_{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 Φ_{min} (resp. Φ_{max}) is true.
- M_I is a tuple of sets of *inheritance modifiers* (μ_{add}, μ_{del}). The set μ_{add} (resp. μ_{del}) consists of tuples $\langle C_{sub}, C_{super}, \Phi_{add} \rangle$ adding (resp. deleting) a superclass C_{super} to C_{sub}, when the first-order formula Φ_{add} (resp. Φ_{del}) is true.

The *150MM* collects all elements appearing in the meta-models of the MMPL. Its elements are decorated with PCs over the features F in FM. An element becomes present in a meta-model when its PC evaluates to true. In addition, we define a *derived presence condition* for the existence of an inheritance path between two classes:

$$\Phi_{inherits(C_j,C_k)} = \bigvee_{f \in paths(C_j,C_k)} \bigwedge_{(C_i,C_{i'}) \in f} \Phi_{(C_i,C_{i'})}$$

where $paths(C_j, C_k)$ is the set of paths between class C_j and C_k in the extended inheritance relation $I \cup \{(C_{sub}, C_{super}) | \langle C_{sub}, C_{super}, \Phi_{add} \rangle \in \mu_{add}\}$ (i.e., the inheritance relation I in the *150MM* plus all inheritance links added by μ_{add}), and $\Phi_{(C_i,C_{i'})}$ is the PC of the inheritance link between C_i and $C_{i'}$ computed as follows:

$$\Phi_{(C_i,C_{i'})} = \begin{cases} \Phi_{add}, \text{if } (C_i, C_{i'}, \Phi_{add}) \in \mu_{add} \\ \neg\Phi_{del}, \text{if } (C_i, C_{i'}) \in I \text{ and } (C_i, C_{i'}, \Phi_{del}) \in \mu_{del} \\ \text{true, if } (C_i, C_{i'}) \in I \text{ and } (C_i, C_{i'}, \Phi_{del}) \notin \mu_{del} \\ \text{false, otherwise} \end{cases}$$

The first case for $\Phi_{(C_i,C_{i'})}$ occurs when an inheritance link is added by an inheritance modifier; the second and third cases occur when an inheritance link in the *150MM* is either deleted or not deleted by a modifier; the fourth case occurs when no inheritance link between C_i and $C_{i'}$ is present.

Remark. Def. 4 uses *modifiers* rather than PCs to change a *characteristic* of an element. Modifiers are not strictly necessary, as they can be emulated with PCs. E.g., given a modifier of the cardinality of a field when Φ_{min} is true, it can be represented as two fields equally named, one with a PC $\neg\Phi_{min}$ and the default cardinality, and the other with a PC Φ_{min} and the modifier cardinality. However, for practical reasons, we favour the use of modifiers to be able to reuse existing meta-model editors to build *150MMs*. These editors expect meta-models with well-formed structure, but are more permissive w.r.t. the OCL invariants, typically handled as uninterpreted strings. Without modifiers, we would need editors to accept classes declaring identically named fields with different PCs (as in the aforementioned example) as well as inheritance cycles. Instead, our approach permits attaching any number

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

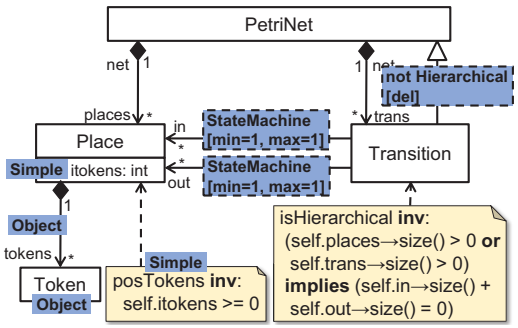


Figure 3. *150MM* annotated with PCs and modifiers.

of modifiers to classes and fields, e.g., as annotations, which is supported by widely used meta-model editors (e.g., the Ecore tree editor [43], Xcore [50] or EMFatic [16]). As modifiers can change the inheritance relations, the *150MM* may include invariants referring to fields that become inherited when a modifier is triggered. Hence, *150MMs* need to have well-formed structure but they may not be well-formed.

Example. Fig. 3 shows the *150MM* for the running example. It is decorated with PCs (blue boxes on top of classes, fields and invariants) and modifiers (dashed blue boxes on top of inheritance relations and references). For example, attribute Place.itokens and invariant posTokens will be present when the PC Simple is true, while the inheritance relation from Transition to PetriNet will be deleted (i.e., it will not appear) when ¬Hierarchical is satisfied. References Transition.in and Transition.out have two cardinality modifiers each; these make the references mono-valued (i.e., with the max cardinality equal to one) when formula StateMachine is true. Note that this *150MM* has intentional errors, which will be uncovered with the techniques we propose in the following sections.

Def. 5 (Feature configuration [38]) A *valid feature configuration* ρ of a product line *MMPL* with feature model $FM = (F, \phi_{MMPL})$ is a subset of its features satisfying ϕ_{MMPL}, i.e., ϕ_{MMPL} evaluates to true when each variable $f \in \phi_{MMPL}$ is substituted by true when $f \in \rho$, and false otherwise. We use $P = \{\rho_i\}_{i \in I}$ for the set of all valid configurations.

Example. Fig. 2 admits eight configurations: $P = \{\langle$Simple$\rangle$, $\langle$Object$\rangle$, $\langle$Simple, StateMachine$\rangle$, $\langle$Simple, Hierarchical$\rangle$, $\langle$Simple, StateMachine, Hierarchical$\rangle$, $\langle$Object, StateMachine$\rangle$, $\langle$Object, Hierarchical$\rangle$, $\langle$Object, StateMachine, Hierarchical$\rangle\}$ (only the leaf features are shown).

Def. 6 (Meta-model derivation) Meta-model MM_ρ is *derived* from a product line *MMPL* using configuration $\rho \in P$ if: MM_ρ contains exactly those elements from the *150MM* whose PCs are satisfied for the features in ρ; the cardinality of fields is overridden according to modifiers μ_{min} and μ_{max} when their formula is true; and inheritance links are added or deleted according to modifiers μ_{add} and μ_{del} when their formula is true. 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 $\rho_0 = \langle$Simple$\rangle$ (Fig. 1(a)), $\rho_1 = \langle$Object$\rangle$ (Fig. 1(b)), $\rho_2 = \langle$Object, Hierarchical$\rangle$ (Fig. 1(c)), and $\rho_3 = \langle$Simple, StateMachine$\rangle$ (Fig. 1(d)).

Def. 4 requires the *150MM* to have a well-formed structure, but not necessarily be well-formed. However, a reasonable requirement of an MMPL is that every derived meta-model is well-formed. We call these MMPLs "well-formed".

Def. 7 (Well-formed MMPL) A product line *MMPL* is *well-formed* iff every $MM \in Prod(MMPL)$ is well-formed.

3 Q1: Ensuring MMPL Well-Formedness

A naive approach to checking well-formedness of an MMPL would generate each $MM \in Prod(MMPL)$ and check well-formedness individually. However, that would be costly and hardly scalable. Hence, this section shows how to check well-formedness of an MMPL without deriving each meta-model.

According to Def. 3, to show that all meta-models of an MMPL are well-formed, we must show that each has (1) a well-formed structure and (2) syntactically correct invariants. We address each of these conditions with a separate analysis.

3.1 Well-Formed Structure

To ensure that the product meta-models of an MMPL have a well-formed structure, we need to check that the four conditions in Def. 3 hold for each product. We show how to check these conditions without enumerating all meta-models, but formulating the corresponding conditions that can be checked on the MMPL.

(1) Every field is owned by one class. Each field is owned by one class in the *150MM*, but this ownership relation will be broken in a product meta-model if the PC of the field is true while the PC of its owning class is false. By Def. 4, the PC of a field f is stronger than the one of its owning class C (i.e., $\Phi_f \Rightarrow \Phi_C$). This ensures that in every configuration where the field is present, the owning class is present too, and so this condition holds by construction for all product meta-models.

(2) Every reference points to a class. A reference r from class C_i to C_j in the *150MM* will become disconnected from C_j in a product meta-model if the PC of r is weaker than the PC of C_j (i.e., $\Phi_r \not\Rightarrow \Phi_{C_j}$). This can be checked for all products by ensuring that for each reference r, $UNSAT(\phi_{MMPL} \wedge \Phi_r \wedge \neg\Phi_{C_j})$, where ϕ_{MMPL} is the formula of the feature model (cf. Def. 1). The predicate *UNSAT* holds if the formula is not satisfiable, and so this ensures no meta-model product contains r but not C_j.

(3) Uniquely determined cardinality and inheritance. A field has unique min and max values in the *150MM*, but cardinality modifiers can change these values. Hence, we have to ensure that the values after applying the modifiers are also unique, i.e., no two modifiers whose condition is true for the same configuration assign different min (or

max) values to the same field:

$$\forall\langle f, m_1, \Phi_{min_1}\rangle, \langle f, m_2, \Phi_{min_2}\rangle \in \mu_{min}\bullet$$
$$m_1 \neq m_2 \implies \neg(\Phi_{min_1} \wedge \Phi_{min_2})$$

This is checked by ensuring that for each two min modifiers over a field, $UNSAT(\phi_{MMPL} \wedge \Phi_{min_1} \wedge \Phi_{min_2})$ (and similarly for max modifiers).

Regarding inheritance, we need to ensure that a class has no conflicting add and del inheritance modifiers:

$$\forall\langle C_{sub}, C_{super}, \Phi_{add}\rangle \in \mu_{add}, \forall\langle C_{sub}, C_{super}, \Phi_{del}\rangle \in \mu_{del}\bullet$$
$$\neg(\Phi_{add} \wedge \Phi_{del})$$

(4) No inheritance cycles. To ensure no meta-model has cycles, we need to show $UNSAT(\phi_{MMPL} \wedge \Phi_{inherits(C_i,C_i)})$ for all classes C_i of the *150MM*, where $\Phi_{inherits(C_i,C_i)}$ is the derived PC of the inheritance paths starting and ending in class C_i (cf. Sec. 2).

3.2 Syntactically Correct Invariants

To ensure that an invariant in the *150MM* is syntactically correct in every product meta-model, we must show that: (1) the PC of each element used by the invariant is compatible with the PC of the invariant, and (2) operators are applied on fields with appropriate cardinality. The following two conditions check this:

(1) Stronger implicit PC. An invariant *wc* in the *150MM* will be syntactically incorrect in some product meta-model if its PC Φ_{wc} is weaker than the PC of some field or class accessed by *wc*. For example, the invariant self.itokens >= 0 contains an access to field itokens, while the OCL expressions Token.allInstances() and self.oclIsKindOf(Token) have a class access each (both to class Token).

Given a field access $o.f$, f is *available for use by* o if the class of o or one of its superclasses own f. Hence, we define the PC of the field access $o.f$, $\Phi_{o.f}$, as follows:

$$\Phi_{o.f} = \begin{cases} \Phi_f, \text{ if } o.class = f.owner \\ \Phi_f \wedge \Phi_{inherits(o.class,\, f.owner)}, \text{ otherwise} \end{cases}$$

with $o.class$ denoting the class of object o, and $f.owner$ denoting the class owning f. We build sets $acc_f(wc)$ and $acc_{cn}(wc)$ of field and class accesses within *wc*, and conjoin their PCs to form the *implicit PC* of *wc*:

$$\varphi_{wc} = \bigwedge_{o.f_i \in acc_f(wc)} \Phi_{o.f_i} \wedge \bigwedge_{C_j \in acc_{cn}(wc)} \Phi_{C_j}$$

Then, the syntactic correctness of *wc* in all product meta-models requires that $\Phi_{wc} \Rightarrow \varphi_{wc}$ holds.

(2) Correct operation typing. A cardinality modifier may turn a multi-valued field (i.e., with max > 1) into a mono-valued one. In that case, an invariant that applies an iterator (e.g., exists) or collection operator (e.g., includes) over the multi-valued field will be incorrect in any product where the field becomes mono-valued. To check this condition, we identify the fields to which an iterator or collection operator is applied, and collect their cardinality modifiers that change max to 1. If one of such modifiers can be true at the same time as the PC of the invariant, there is an error. Hence, let $acc_{mv}(wc)$ be the set of fields to which invariant *wc* applies a collection operator. We need to ensure the following condition for the modifiers that change max of the fields to 1:

$$UNSAT(\phi_{MMPL} \wedge \Phi_{wc} \wedge \bigvee_{\substack{\langle f, 1, \Phi_{max_j}\rangle \in \mu_{max} \\ \text{s.t. } f \in acc_{mv}(wc)}} \Phi_{max_j})$$

Conversely, mono-valued fields may become multi-valued due to modifiers. Thus, we identify all operators over mono-valued fields (e.g., + or <) and check the modifiers that can turn them into collections, leading to an error. This check is similar to the one above, but considers modifiers $\langle f, n, \Phi_{max}\rangle$ with $n > 1$ or $n = \infty$.

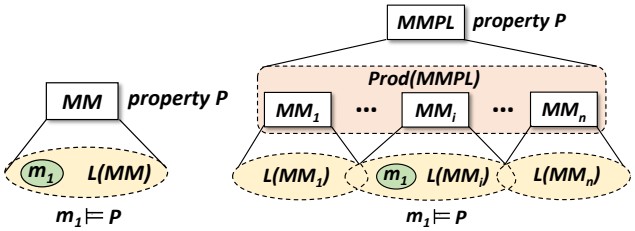
Example. We analyse the syntactic correctness of invariant isHierarchical in Fig. 3, in all product meta-models. Its set of field accesses is acc_f(isHierarchical) = {self.places, self.trans, self.in, self.out}, where self is typed by class Transition. The PC of the first two field accesses is $\Phi_{self.places} = \Phi_{self.trans}$ = Hierarchical, while the PC of the last two is $\Phi_{self.in} = \Phi_{self.out}$ = true. Hence, the invariant has the implicit PC $\varphi_{isHierarchical}$ = Hierarchical, while it declares the PC $\Phi_{isHierarchical}$ = true in the *150MM*. We require $\Phi_{isHierarchical} \Rightarrow \varphi_{isHierarchical}$, but true $\not\Rightarrow$ Hierarchical, hence the first syntactic correctness condition does not hold. This means that the invariant appears in meta-models that do not set the feature Hierarchical to true, in which case it has syntax errors because Transition will lack fields places and trans. We fix the problem by setting the PC of invariant isHierarchical to Hierarchical in the *150MM*.

To analyse operation typing, we compute the fields to which a collection operator is applied: acc_{mv}(isHierarchical) = {places, trans, in, out}. Both in and out have modifiers changing max to 1 when StateMachine is true. Assuming the invariant has PC Hierarchical, the collection operator size() is applied over mono-valued fields on configurations where StateMachine ∧ Hierarchical is true. However, in→size() is a shorthand for in→asSet()→size(), which converts in into a set and permits applying operator size() to the result. Therefore, the invariant is correctly typed in all products.

4 Q2: Analysing Instance Properties

Guaranteeing syntactic correctness is not sufficient for ensuring a valid meta-model, as we need to assure that the set of accepted models is the one the designer intends. Given a meta-model *MM*, we write *L*(*MM*) to refer to the (possibly infinite) set of models that are correctly typed by *MM* and satisfy its invariants: its language. To validate *MM*, we need to check that *L*(*MM*) does not contain models considered invalid, or is missing models considered valid.

Fig. 4(a) shows how this validation process is normally automated using constraint solving [6, 28, 42] and model



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

Figure 4. Analysis of meta-model instance properties.

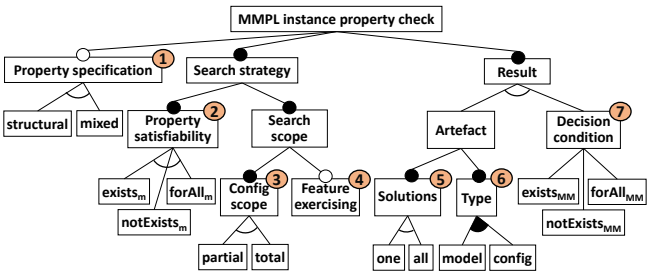


Figure 5. Options for MMPL instance property analysis.

finders [6, 19, 21, 26]. A *model finder* is a tool that receives a meta-model (including its OCL invariants) and a set of model properties as input, and uses constraint solving to find a model that conforms to the meta-model and exhibits the specified properties. Model finders used to validate meta-models, like the USE Validator [26] and EMFtoCSP [19], permit expressing the desired model properties in OCL.

To check a property on the instances of a meta-model, the engineer expresses the property P using OCL, and uses a model finder to look for a model satisfying P within the search bounds, or determine there are none. Checking if 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 if a valid model exists (i.e., whether $L(MM)$ is not empty). This ensures that MM has no conflicting invariants.

Since the analysis of instance properties is a useful and accepted meta-model validation technique, we propose lifting it to MMPLs, as Fig. 4(b) shows. In Sec. 4.1, we classify the relevant property types, and in Sec. 4.2, we propose a method to analyse those properties at the product line level.

4.1 A Classification of Property Types

We begin by describing the different options for specifying properties on instances of meta-models of an MMPL. They are depicted as a feature model in Fig. 5.

(1) Property specification. A property P may need to refer not only to structural elements of the meta-models, but also to features of the feature model. We call such properties *mixed*. For example, the following property checks whether any model that only contains transitions with one input place belongs to configurations where feature StateMachine is true: Transition.allInstances()→ **forAll**(p | p.in→size() = 1) **implies** StateMachine.

(2) Property satisfiability. Given a property P, we may want to analyse whether some meta-model MM in the MMPL has at least one model $m \in L(MM)$ satisfying P, whether it has no model satisfying P, or whether all its models satisfy P. These options correspond to features exists_m, notExists_m and forAll_m in Fig. 5.

(3) Configuration scope. A property 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 P. 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 attribute itokens is only present when this feature is true.

(4) Feature exercising. To illustrate the meta-model elements a configuration makes available, to compare configurations, or to reason about feature interactions, it is useful to produce models that contain instances of the elements added due to the selected features. For example, using this option, models of configurations that select feature Object need to contain at least one instance of Token, as this class is only added when this feature is true.

(5) Solutions. The result of the analysis may be a model of *some* meta-model in the MMPL satisfying P, or a model of *every* meta-model in the MMPL satisfying P. These options correspond to features one and all in Fig. 5.

(6) Result type. In addition to retrieving a model m satisfying P, the analysis may return the feature configuration that produces the meta-model of which m is an instance.

(7) Decision condition. 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 result in this case is a yes/no answer to each question.

This space of options enables us to express many interesting properties of the MMPL, as shown in Table 1. The table also includes the (partial) configuration that corresponds to the given property type, and an example.

Analyses 1–3 study the instantiability (i.e., the existence of models) of the meta-models in an MMPL. While this is a basic correctness condition, MMPL validation may require stronger conditions, like discovering some/all meta-models without instances (Analysis 4). If a meta-model is not instantiable, it may indicate that the configuration used to create it contains incompatible features. Actually, 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. To analyse this, we propose using feature-exercising instance generation to produce models

Table 1. Types of instance properties and analyses for MMPLs (partial configurations indicated with ...).

	Analysis	Description	Configuration (cf. Fig. 5)	Example (P=property; R=result)
1	MMPL instantiability	Finds 1 instantiable configuration	⟨one, exists$_m$, total, config⟩	**R:** An instantiable configuration: ⟨Object, Hierarchical⟩
2	MMPL instantiability witness	Finds 1 model of 1 instantiable MM	⟨one, exists$_m$, total, model⟩	**R:** A model with a PetriNet and a Place objects
3	Instance generation	Finds 1 model of each instantiable MM	⟨all, exists$_m$, total, model, config⟩	**R:** Eight example models are populated
4	Vacuous meta-model discovery	Finds all non-instantiable configurations	⟨all, notExists$_m$, total, config⟩	**R:** Empty set: all configurations are instantiable
5	Vacuous feature combinations	Finds all configurations where no model uses every meta-model element present due to feature selections	⟨all, notExists$_m$, total, feature exercising, config⟩	**R:** Two configurations: ⟨Simple, StateMachine, Hierarchical⟩, and ⟨Object, StateMachine, Hierarchical⟩
6	Global weak property	Checks if a property is satisfied by some model of every meta-model in the MMPL	⟨forAll$_{MM}$, exists$_m$, ...⟩	**P:** all meta-models admit models with more transitions than places; **R:** false (StateMachines do not) Transition.allInstances()→**size**() > Place.allInstances()→**size**()
7	Local weak property	Checks if a property is satisfied by some model of some meta-model in the MMPL	⟨exists$_{MM}$, exists$_m$, ...⟩	**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	Global invariant	Checks if a property is satisfied by every model of every meta-model in the MMPL	⟨forAll$_{MM}$, forAll$_m$, ...⟩	**P:** all models have at least one place; **R:** false Place.allInstances()→**notEmpty**()
9	Local invariant	Checks if a property is satisfied by every model of some meta-model in the MMPL	⟨exists$_{MM}$, forAll$_m$, ...⟩	**P:** some meta-models only admit models with more places than transitions; **R:** false Place.allInstances()→**size**() > Transition.allInstances()→**size**()
10	Global safety property	Checks if no model of the MMPL satisfies a property	⟨forAll$_{MM}$, notExists$_m$, ...⟩	**P:** no model has isolated transitions; **R:** false Transition.allInstances()→**exists**(t | t.in→**isEmpty**() **and** t.out→**isEmpty**())
11	Local safety property	Checks if some meta-model of the MMPL has no model satisfying a property	⟨exists$_{MM}$, notExists$_m$, ...⟩	**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 property	A property mixing meta-model elements and features	⟨mixed, ...⟩	**P:** transitions with one input are only on StateMachines; **R:** false Transition.allInstances()→**forAll**(p | p.in→**size**()=1) **implies** StateMachine

with instances of the elements activated by the given configuration. Meta-models where no such instances exist typically reveal a feature conflict (Analysis 5).

Example. To illustrate the usefulness of feature-exercising instance generation, and the subtle problems it can detect, consider the following scenario. A model that exercises feature Hierarchical should contain hierarchical transitions (i.e., with non-empty collections places and trans). However, if the configuration also selects feature StateMachine, the resulting meta-model does not accept models with hierarchical transitions due to the invariant isHierarchical. This shows that features Hierarchy and StateMachine are incompatible, and would be discovered by Analysis 5. This problem can be solved by adding a modifier that sets the min cardinality of references in and out to 0 when both features StateMachine and Hierarchy are selected.

More specific analyses focus on the satisfiability of properties. Depending on the number of models satisfying a property, we say the property is *weak* if some model satisfies the property, *invariant* if all models of a given meta-model satisfy the property, and *safety* if no model satisfies it. Additionally, we say a weak property is *global* if it is a weak property for every meta-model in the scope; while it is *local* if it is a weak property for some meta-model. This global/local classification also holds for invariants and safety properties. Rows 6–11 in Table 1 show the configuration for these property types, as well as examples formulated both verbally (to facilitate comprehension) and in OCL (to automate validation).

Overall, our novel classification defines different kinds of lifted instance analyses for MMPLs. Moreover, it also defines new specification possibilities like mixed properties, with no counterpart in the analysis of plain meta-models.

4.2 Lifting Instance Analysis to MMPLs

As explained at the beginning of Sec. 4, checking a property on the instances of a meta-model amounts to finding a model that conforms to the meta-model and satisfies the property. This search is typically performed using model finders. However, the number of meta-models in an MMPL may grow exponentially, and hence, performing model finding over

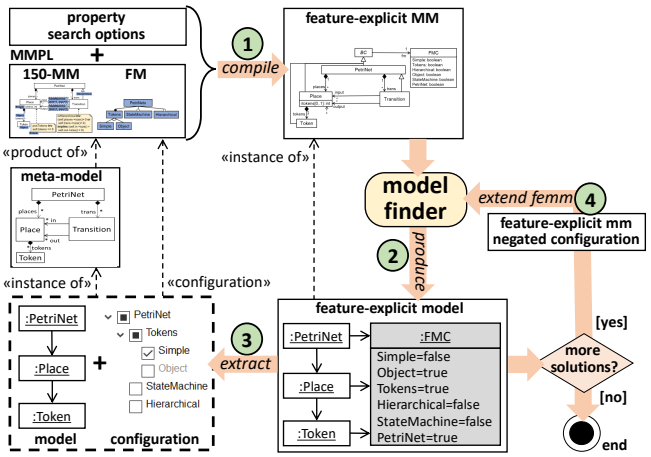


Figure 6. Workflow of instance property analysis.

each single meta-model may become very time-consuming. For this reason, we lift the analysis over an extended version of the *150MM* that contains the meta-model elements introduced by every variant and emulates the feature configurations, presence conditions and modifiers using invariants. We call this meta-model *feature explicit meta-model* (FEMM). This way, the problem of checking a property on some/every meta-model of the MMPL is recasted to finding one or more instances of the FEMM. Fig. 6 illustrates the proposed workflow to analyse the properties introduced in Sec. 4.1.

In Step 1, we *compile* the *150MM*, the feature model, and the property of interest into a FEMM. This extends the *150MM* with invariants that emulate the semantics of the PCs and modifiers, an extra class FMC with a boolean attribute for each feature in the feature model, an invariant stating the allowed configurations, and the property to check. Fig. 7 shows the FEMM for the running example.

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 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.

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 those already found. For this purpose, the invariant forbids objects of type FMC to take the same attribute values as the configurations found so far. Then, a new search is performed.

In the following, we explain the steps to build the FEMM, and how to perform the analysis of the different property types. Fig. 7 shows the resulting FEMM for the running example, and Listing 1 provides a high-level description of the algorithm to construct it.

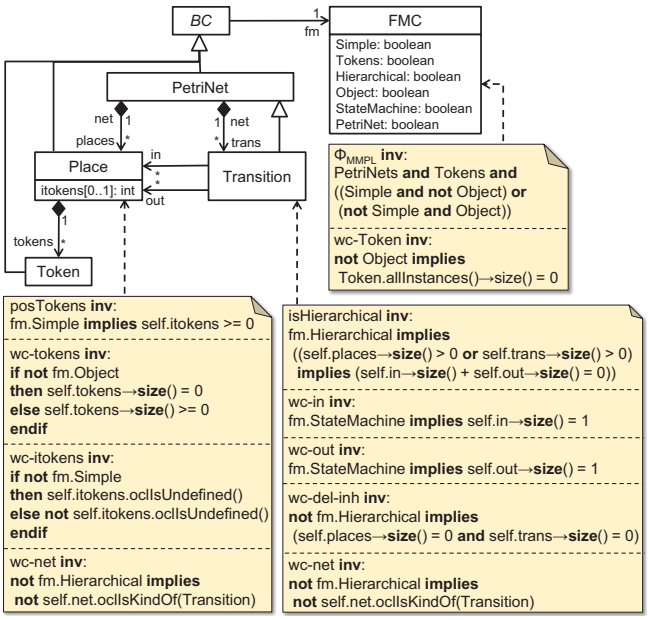


Figure 7. FEMM derived from the MMPL in Figs. 2 and 3.

4.2.1 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). In addition, it embeds the feature model, 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 ϕ_{MMPL} of class FMC in Fig. 7). To make this class accessible from any other, we create a base class BC for the rest of classes, and add a reference from this base class to the class representing the feature model. This way, every class has access to the attributes in FMC, i.e., to the feature configuration. Lines 6–12 of Listing 1 take care of embedding the feature model.

4.2.2 Emulating the PCs PCs are emulated by extra invariants in the FEMM, built according to lines 15–22 of Listing 1.

First, a PC in a class is converted into an invariant of FMC which ensures that there are no instances of the class when the PC is not true. As an example, class FMC in Fig. 7 declares the invariant wc-Token to represent the PC of class Token.

If the PC affects an invariant, the latter is rewritten so that it is only checked when the condition is met (see invariant posTokens in class Place, where the PC Simple is rewritten as fm.Simple because the feature values are stored in class FMC).

Finally, a PC in a field is represented as an invariant which ensures that the field is empty (in case of references) or undefined (in case of attributes) when the PC is false. This invariant is added to the field's owner class. Moreover, the min cardinality of the field is set to 0, as otherwise, the field could never be empty. As an example, invariant wc-itokens in class Place is derived from the PC in attribute Place.tokens. The min cardinality of the attribute, which was 1 in the *150MM*, becomes 0. This way, 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.

4.2.3 Emulating the Modifiers Cardinality modifiers are translated into invariants attached to the field's owner class. Such invariants demand the field to satisfy the modified cardinality when the modifier condition is true, or the cardinality specified in the *150MM* otherwise (see lines 25–29 in Listing 1). For example, invariants wc-in and wc-out in class Transition are derived from modifiers in references Transition.in and Transition.out. In addition, the min cardinality of the field is set to the lowest among 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 highest among its value in the *150MM* and its max cardinality modifiers. In the example, Transition.in and Transition.out do not change their cardinality as they originally had the min and max possible values ([0..*]).

Inheritance modifiers are handled in lines 30–35 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 have no value when the modifier condition is true (resp. false). In both cases, the inheritance link is added to the FEMM. As an example, invariant wc-del-inh in class Transition is derived from the inheritance modifier in the *150MM*. 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 the incoming reference net to the superclass PetriNet (the invariants are specific to monovalued references, which for space constraints are not considered in Listing 1).

4.2.4 Exercising Features Feature exercising allows illustrating the meta-model elements specific to the features selected in a configuration. If this analysis option is chosen, we augment class FMC with invariants enforcing the existence of instances of the classes and fields activated by the selected features. A PC on a class becomes an invariant requiring at least one object of the class when the PC is true. A PC or modifier on a field is translated into an invariant requiring that the field has non-empty value in some object when the presence or modifier condition is true. 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 (lines 38–44 in Listing 1).

4.2.5 Embedding the Property The property to be analysed and the configuration scope (when partial) are added as invariants of class FMC. Even if the property is mixed, it can be embedded without changes because class FMC defines the feature values as attributes, so they are accessible from the added invariant. If it is a global property (i.e., we aim to check whether it is satisfiable for all instances of a meta-model, see Table 1), then the property is negated.

```
1  buildFeatureExplicitMM (150MM : in, FM : in, FEMM : out)
2    create FEMM = new meta-model
3    add classes, fields, invariants and inheritance in 150MM to FEMM
4
5    -- 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 FMC to BC, with cardinality [1,1]
9    for each feature f in FM, add boolean attribute f to FMC
10   add invariant to FMC = formula in FM
11   add BC as superclass to all classes with no parent class
12   add FMC, BC and fm to FEMM
13
14   -- emulate presence conditions
15   for each class c with presence condition pc:
16     add invariant to FMC = not pc implies size(c) = 0
17   for each invariant wc with presence condition pc:
18     set wc = pc implies wc
19   for each field f with presence condition pc:
20     set def = min-cardinality(f)
21     add invariant to owner(f) = if not pc then size(f) = 0 else def endif
22     set min-cardinality(f) = 0
23
24   -- emulate modifiers (max cardinality modifiers omitted for brevity)
25   for each field f with cardinality modifier m and condition mc:
26     set def = min-cardinality(f)
27     add invariant to owner(f) = if mc then size(f)>=m
28                                 else size(f)>=def endif
29     set min-cardinality(f) = minimum(def, min)
30   for each inheritance relation inh with del/add inheritance modifier:
31     for each field f inherited by inh:
32       add invariant to subclass = pc/not pc implies size(f) = 0
33     for each reference r to superclass or ancestor:
34       add invariant to owner(r) = pc/not pc implies
35                                   not r.exists(o | o.oclIsKindOf(subclass))
36
37   -- exercise features
38   for each class c with presence condition pc:
39     add invariant to FMC = pc implies size(c) > 0
40   for each field f with presence condition or modifier pc:
41     add invariant to FMC = pc implies size(o.f) > 0 for some o
42   for each inheritance relation inh with del/add inheritance modifier:
43     for each field f inherited by inh:
44       add invariant to FMC = not pc/pc implies size(o.f)>0 for some o
45
46   return FEMM
```

Listing 1. Simplified algorithm to build the FEMM.

4.2.6 Analysing the Meta-Model Once the FEMM has been constructed, we look for an instance of it using a model finder. We require such an instance to contain exactly one object of type FMC, which will hold a valid configuration.

To check MMPL instantiability, instance generation with or without feature exercising, and weak property satisfaction (feature exists_m in Fig. 5), the finder is invoked 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, the actual model and configuration (i.e., the attribute values in the FMC object) must be extracted from it, while the model's meta-model can be produced from the configuration. This yields a configuration, a corresponding meta-model, and an instance model satisfying the property.

Example. Fig. 6 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). Further solutions can be found by extending the FEMM with an invariant that forbids the found configurations as solutions, and invoking the finder again. For example, in Fig. 6, this would amount to adding the following invariant to class FMC: not (not Simple and Object and Tokens and not Hierarchical and not StateMachine and PetriNet). This process can be iterated until the finder does not return further solutions.

To analyse vacuous meta-model discovery and safety properties (feature $\mathsf{notExists}_m$), the model finder is invoked iteratively, each time forbidding the configurations previously found. When the finder does not find new results, the analysis returns the remaining valid configurations that were not found by the solver. These configurations identify the meta-models which have no instances satisfying the property.

Finally, checking global invariants (feature forAll_m) follows the same process as for safety properties. The only difference is that, as explained in Sec. 4.2.5, the property is negated. Moreover, the set of returned configurations identifies the meta-models with all their instances satisfying the property.

5 Tool Support

We have realized our approach in an Eclipse plugin called Merlin, freely available at http://miso.es/tools/merlin. Merlin extends FeatureIDE [30] to create feature models, handle configurations and build products. It integrates the USE Validator [26] for instance property analysis, EMF for describing meta-models [43], the Eclipse OCL project [15] for static analysis of OCL expressions, and Sat4J [4] for the constraint solving needed by the static analysis.

Fig. 8 shows a screenshot of the tool being used to validate an instance property on the running example. 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 feature model is built with FeatureIDE (label 2), and the *150MM* is created using a meta-model editor (OCLinEcore in Fig. 8, with label 3). We provide a set of annotations to define modifiers and PCs within Ecore-based *150MM*s.

The tool supports all syntactic and property analyses we have presented, and displays results in the Eclipse problem view (label 4). Furthermore, it allows checking the satisfiability of PCs and the applicability of modifiers. This analysis performs SAT solving to discover unsatisfiable PCs and modifier conditions, taking into account the formula implied by

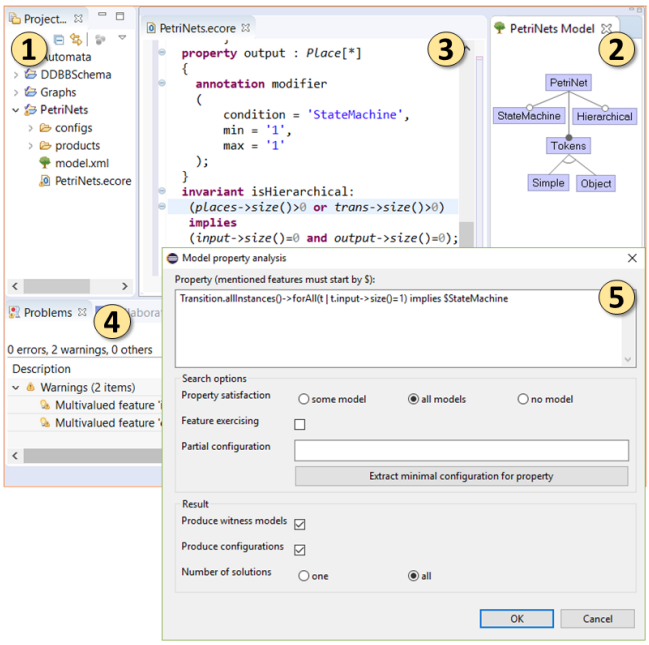


Figure 8. Merlin in action.

the feature model. For example, a PC Simple ∧ Object is not satisfiable, being reported as an error.

Label 5 in the figure shows the wizard for the instance property analysis. Here the user can enter structural and mixed properties, and specify all the different search and result configuration options. Given a property, the tool can extract its implied PC, which can be used to reduce the search scope. Internally, Merlin uses the USE Validator for model finding, and the results are parsed back into EMF models and FeatureIDE configuration files.

6 Evaluation

This section evaluates the scalability of our lifted analyses, compared with checking an explicit enumeration of all meta-models in the MMPL. Sec. 6.1 evaluates the syntactic analysis, and Sec. 6.2 evaluates the instance property analysis.

6.1 Efficiency of Syntactic Analysis

In this section, we compare the performance of the lifted syntactic analysis presented in Sec. 3, with respect to checking every meta-model of the MMPL. The latter requires generating every meta-model, validating its structure, and type-checking its invariants.

For this evaluation, we considered 5 MMPLs of different complexities, ranging from 6 to 48 features. Two were created by us (the running example, Automata); one was created from a set of existing meta-models (Relational DDBB); and the other two were taken or adapted from the literature (Graphs, Role modelling). The Graphs product line is a common benchmark in the SPL community [29], but as we focus on the language and not on the algorithms, we included features

Table 2. Lifted and enumeration-based syntactic analyses.

Name	#Feats	#MMs	#classes/#invs /#PCs/#modifs	Lifted time	Enum time
Running example	6	8	4/2/5/2	0,039s	0,19s
Relational DDBB	10	24	7/0/17/0	0,094s	0,45s
Graphs [29]	16	256	5/6/14/3	0,103s	22,36s
Automata	20	2.016	6/5/18/0	0,135s	102,9s
Role modelling [27]	48	>2.395.000	40/0/32/9	0,735s	>1h

related to its structure in order to add some OCL invariants. The MMPL Relational DDBB was manually created by us, from 9 existing meta-models built by third parties, available in the ATL meta-model zoo[2]. All MMPLs used in these experiments are available at http://miso.es/tools/merlin.

The experiment was made on a Windows 10 computer, with i7-6500U processor and 16Gb of memory. We repeated each analysis 5 times both at the MMPL level and using enumeration, taking averages. Table 2 summarizes the results.

Our first conclusion is that lifted analysis scales well, even to large MMPLs (<1s), while the enumerative analysis does not scale (we interrupted the analysis of the Role modelling MMPL after 1 hour). We have to stress that, while in the enumerative approach we checked only well-formedness of each meta-model (both structure and OCL), in our lifted analysis we *also* checked for unused features, PC satisfiability and modifier applicability. Not only is lifted analysis faster, but it also makes it easier to provide valuable feedback to the designer. First, by discovering unsatisfiable PCs and non-applicable modifiers, we can point to elements in the *150MM* that will never appear in any meta-model. This analysis would be costly to perform on the meta-model level. Second, if the MMPL is not well-formed, the enumerative approach produces wrong meta-models which cannot be persisted using EMF. This complicates generating useful feedback and diagnosing the error cause.

Threats to validity. Although the experiment shows much better scalability of lifted analysis, a more extensive study with increasing number of modifiers, PCs and invariants would be needed to better evaluate scalability issues.

6.2 Efficiency of Instance Property Analysis

Our second experiment compares the lifted instance property analysis presented in Sec. 4, with respect to an enumerative approach. As a representative case of property type, we consider the analysis of MMPL instantiability, i.e., finding a configuration that yields an instantiable meta-model. Using an enumerative approach, this implies generating one meta-model of the MMPL at a time, checking its instantiability, and finishing if it has instances. The sooner an instantiable meta-model is found, the faster the analysis. This means the analysis time in the enumerative approach depends on the percentage of instantiable meta-models in the MMPL: if

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

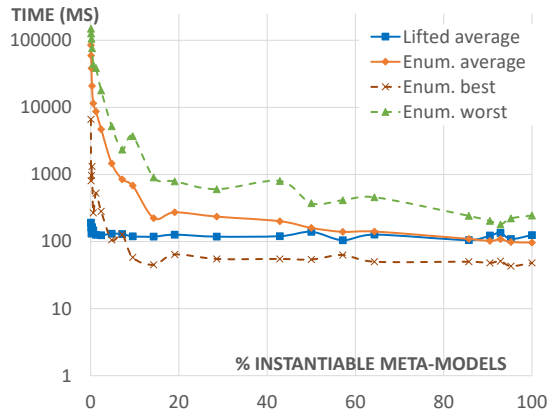


Figure 9. Lifted and enumeration-based property analyses.

many meta-models are instantiable, the likelihood of finding one soon increases. For this reason, this experiment considers MMPLs with different instantiability ratios.

Specifically, we used for the experiment the Automata MMPL as it produces a large number of meta-models with invariants (see Table 2). While all 2016 meta-models in this MMPL have instances, we manually built 22 modified 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 2016, another had two, and so on). This was done to emulate MMPLs with different meta-model instantiability ratios. Then, we used both approaches (lifted and enumerative) to analyse MMPL instantiability of each version, 40 times each, computing the average time. In the enumerative approach, the traversal of the meta-models in the MMPL was randomized in each analysis.

Fig. 9 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 enumerative approach, but not for the lifted one because they are similar to the average.

The average time of the lifted analysis is lower than the enumerative one when the percentage of instantiable meta-models is up to 85%, and when this percentage is under 10%, the lifted analysis is up to 1.000x faster. When more than 85% meta-models are instantiable, the performance of the lifted analysis is only slightly lower but still reasonable (100 vs 120 ms). The best case for the enumerative approach corresponds to finding an instantiable meta-model at the first attempt, in which case it is up to 3x faster than the lifted analysis because the constraint solving problem is easier. So, the lifted analysis is only slightly slower than the best case of the enumerated approach (and always below 150 ms), but it is several orders of magnitude faster than the corresponding worst case, hence confirming its benefits w.r.t. the enumerative approach.

Threats to validity. While these results provide some evidence of the benefits of lifting the analysis of instance properties to the product line level, a threat to their generality is the low number of artefacts used in the evaluation, and the fact that they have been produced synthetically by hand. Moreover, we have evaluated only MMPL instantiability, but not the other property types.

7 Related Work

MDE is used to support SPLs in various ways [22], like improving traceability of SPLs [1] or expressing feature models as meta-models [18], and there are standardization attempts [9]. Conversely, SPLs allow expressing variability within MDE solutions, as we do in this paper. For example, model-based product lines can be used as an intermediate step towards code generation [47], and some companies (e.g., in defence, embedded and automotive domains) use SPLs to manage the variability across their model assets [39, 48, 51]. These works apply SPLs to specific modelling languages, at the model-level. In contrast, we work at a higher level of abstraction, on product lines of modelling languages.

Many works recognise the need to express variability in modelling languages [7, 31]. The UML specification includes informal but explicit *semantic variation points* to address different usage contexts [8, 44], and approaches to language engineering propose *documenting* semantic variations as feature models [20]. Beyond mere documentation, MMPLs have been applied to improve the reusability of domain-specific modelling languages [49] and their associated transformations [35]. These works lack means to ensure the resulting meta-models are correct. However, building MMPLs can be challenging and error-prone if there are many features or meta-model invariants. This makes necessary analysis techniques like the ones we propose in this paper.

A key ingredient of model-based SPLs is the mechanism to express model variability and automate product derivation. Approaches like delta-modelling [27, 41] or VML* [52] are language-independent and can be applied to SPLs of models or meta-models. Their focus is on expressivity, being possible to apply arbitrary operations to synthesize a product. However, this makes ensuring syntactic and semantic correctness challenging. In contrast, our approach based on PCs and modifiers facilitates analysis, and enables instance property analysis by model finding. Without such techniques, feature incompatibilities (e.g., StateMachine and Hierarchy) may go unnoticed and percolate to the final solution, creating errors.

Many analysis techniques have been proposed for SPLs, such as lifting analyses from single artefacts to the product line level, or analysing a subset of all derivable products, like in pairwise testing (see [46] for a classification and survey). Both syntactic [24, 25] and behavioural analyses (e.g., model checking [14, 45]) have been lifted to SPLs. The advantage of lifting is a better efficiency and scalability with respect to an enumerative approach. In addition, richer specification means may be obtained (e.g., mixed properties).

Just a few analysis techniques have been developed for model-based SPLs. In [11], model-based SPLs are analysed to ensure that each generated model conforms to its meta-model and fulfils its integrity constraints. This corresponds to a syntactic analysis lifting. In our case, we work at the meta-model level, and our lifted analyses ensure that each product meta-model is syntactically well-formed and has the required instantiability properties. Moreover, we define a catalogue of types of instantiability analyses that can be done at the MMPL level. In [3], the authors present the tool Clafer, which unifies meta-models and feature models. It supports the analysis of our MMPL instantiability property by a compilation into Alloy, but not the lifted analysis of any other instance property. In [17], the authors synthesize random, erroneous model-based SPLs for a modelling language, to showcase typical errors in the SPL design.

Altogether, analysis of MMPLs at the product line level are currently lacking. For this reason, we have lifted both syntactic analyses and meta-model validation techniques based on constraint solving [6, 42] to MMPLs. To the best of our knowledge, these liftings, along with the classification of instance property analyses, are novel contributions.

8 Conclusions and Future Work

In this paper, we have argued for the need to express and handle variability within modelling languages. For this purpose, we have proposed a notion of MMPL using an annotative approach extended with modifiers. This approach considers well-formedness constraints and facilitates analysis. We have lifted syntactic and instance property analyses to MMPLs, and proposed a classification for property types. We have implemented the approach in the Merlin tool, and reported on experiments showing the scalability benefits, with respect to an explicit analysis of each product in the MMPL.

We believe that this work opens the door to a wider use of variability within MDE to foster reusability. We are currently working on reusable model transformations, combining this notion of MMPLs with ideas from generic programming [12, 13]. Transformations defined in this manner become product lines themselves, reusable for each meta-model of the MMPL.

We also plan to expand our analyses to discover subsumption of language variants. For example, in the case of Petri nets, all StateMachine nets are FreeChoice nets, but not vice versa [33]. This analysis will help in constraining the feature model to reduce the configuration space and reflect expected language relations. Extending our notion of MMPL with *type modifiers* that change the type of a field is also future work.

Acknowledgements

Work funded by NSERC, the Spanish MINECO (TIN2014-52129-R) and the Madrid region (S2013/ICE-3006).

References

[1] Nicolas Anquetil, Uirá Kulesza, Ralf Mitschke, Ana Moreira, Jean-Claude Royer, Andreas Rummler, and André Sousa. 2010. A Model-Driven Traceability Framework for Software Product Lines. *Software and System Modeling* 9, 4 (01 Sep 2010), 427–451.

[2] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. 2009. Model Superimposition in Software Product Lines. In *Proc. of ICMT'09 (LNCS)*, Vol. 5563. Springer, 4–19.

[3] Kacper Bak, Zinovy Diskin, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2016. Clafer: unifying class and feature modeling. *Software and System Modeling* 15, 3 (2016), 811–845.

[4] Daniel Le Berre and Anne Parrain. 2010. The Sat4j library, release 2.2. *JSAT* 7, 2-3 (2010), 59–6.

[5] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition.* Morgan & Claypool Publishers, San Rafael, California (USA).

[6] Jordi Cabot, Robert Clarisó, and Daniel Riera. 2014. On the Verification of UML/OCL Class Diagrams Using Constraint Programming. *Journal of Systems and Software* 93 (2014), 1–23.

[7] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. 2009. Variability within Modeling Language Definitions. In *Proc. of MDELS'09 (LNCS)*, Vol. 5795. Springer, Berlin, Heidelberg, 670–684.

[8] Franck Chauvel and Jean-Marc Jézéquel. 2005. Code Generation from UML Models with Semantic Variation Points. In *Proc. of MODELS'05 (LNCS)*, Vol. 3713. Springer, 54–68.

[9] CVL. 2012. http://www.omgwiki.org/variability/doku.php.

[10] Krzysztof Czarnecki and Michal Antkiewicz. 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. of GPCE'05 (LNCS)*, Vol. 3676. Springer, 422–437.

[11] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. of GPCE'06.* ACM, New York, NY, USA, 211–220.

[12] Juan de Lara and Esther Guerra. 2013. From Types to Type Requirements: Genericity for Model-Driven Engineering. *Software and System Modeling* 12, 3 (2013), 453–474.

[13] Juan de Lara, Esther Guerra, Marsha Chechik, and Rick Salay. 2018. Model Transformation Product Lines. In *Proc. of MODELS'18.* ACM.

[14] Aleksandar S. Dimovski and Andrzej Wasowski. 2017. Variability-Specific Abstraction Refinement for Family-Based Model Checking. In *Proc. of FASE'17 (LNCS)*, Vol. 10202. Springer, 406–423.

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

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

[17] João Bosco Ferreira Filho, Olivier Barais, Mathieu Acher, Jérôme Le Noir, Axel Legay, and Benoit Baudry. 2015. Generating Counterexamples of Model-Based Software Product Lines. *STTT* 17, 5 (2015), 585–600.

[18] Abel Gómez and Isidro Ramos. 2010. Cardinality-Based Feature Modeling and Model-Driven Engineering: Fitting them Together. In *Proc. of VaMoS'10.* 61–68.

[19] Carlos A. González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. 2012. EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. In *Proc. of FormSERA'12.* IEEE Press, Piscataway, NJ, USA, 44–50.

[20] Hans Grönniger and Bernhard Rumpe. 2010. Modeling Language Variability. In *Proc. of 16th Monterey Workshop on Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems (LNCS)*, Vol. 6662. Springer, Berlin, Heidelberg, 17–32.

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

[22] J.-M. Jezequel. 2012. Model-Driven Engineering for Software Product Lines. *ISRN Software Engineering* 2012, 670803 (2012), 24pp.

[23] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[24] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. 2012. Type Checking Annotation-Based Product Lines. *ACM Trans. Softw. Eng. Methodol.* 21, 3 (2012), 14:1–14:39.

[25] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don S. Batory. 2009. Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach. In *Proc. of TOOLS EUROPE'09.* Springer, Berlin, Heidelberg, 175–194.

[26] Mirco Kuhlmann and Martin Gogolla. 2012. From UML and OCL to Relational Logic and Back. In *Proc. of MODELS'12 (LNCS)*, Vol. 7590. Springer, Berlin, Heidelberg, 415–431.

[27] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. 2014. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Proc. of SLE'14 (LNCS)*, Vol. 8706. Springer, 141–160.

[28] Jesús J. López-Fernández, Esther Guerra, and Juan de Lara. 2015. Example-Based Validation of Domain-Specific Visual Languages. In *Proc. of SLE'15.* ACM, 101–112.

[29] Roberto E. Lopez-Herrejon and Don Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *Generative and Component-Based Software Engineering*, Jan Bosch (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 10–24.

[30] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE.* Springer. See also https://featureide.github.io/.

[31] David Méndez-Acuña, José A. Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. 2016. Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. *J. Computer Languages, Systems & Structures* 46 (2016), 206–235.

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

[33] T. Murata. 1989. Petri Nets: Properties, Analysis and Applications. *Proc. IEEE* 77, 4 (1989), 541–580.

[34] OCL. 2014. http://www.omg.org/spec/OCL/.

[35] Gilles Perrouin, Moussa Amrani, Mathieu Acher, Benoît Combemale, Axel Legay, and Pierre-Yves Schobbens. 2016. Featured Model Types: Towards Systematic Reuse in Modelling Language Engineering. In *Proc. of MiSE@ICSE'16.* ACM, New York, NY, USA, 1–7.

[36] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering. Foundations, Principles and Techniques.* Springer-Verlag Berlin Heidelberg.

[37] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2015. Collaborative Repositories in Model-Driven Engineering. *IEEE Software* 32, 3 (2015), 28–34.

[38] Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, and Marsha Chechik. 2014. Lifting Model Transformations to Product Lines. In *Proc. of ICSE'14.* ACM, New York, NY, USA, 117–128.

[39] Alexander Schlie, David Wille, Sandro Schulze, Loek Cleophas, and Ina Schaefer. 2017. Detecting Variability in MATLAB/Simulink Models: An Industry-Inspired Technique and Its Evaluation. In *Proc. of SPLC'17.* ACM, 215–224.

[40] D C Schmidt. 2006. Guest Editor's Introduction: Model-Driven Engineering. *Computer* 39, 2 (Feb. 2006), 25–31.

[41] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. DeltaEcore – A Model-Based Delta Language Generation Framework. In *Modellierung (LNI)*, Vol. 225. GI, Bonn, 81–96.

[42] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. 2010. Verifying UML/OCL Models Using Boolean Satisfiability. In *Proc. of DATE'10.* IEEE Computer Society, 1341–1344.

[43] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2^{nd} Edition.* Addison-Wesley Professional, Upper Saddle River, NJ.

[44] Ali Taleghani and Joanne M. Atlee. 2006. Semantic Variations Among UML StateMachines. In *Proc. of MODELS'06 (LNCS)*, Vol. 4199. Springer,

245–259.

[45] Maurice H. ter Beek, Erik P. de Vink, and Tim A. C. Willemse. 2017. Family-Based Model Checking with mCRL2. In *Proc. of FASE'17 (LNCS)*, Vol. 10202. Springer, 387–405.

[46] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 6:1–6:45.

[47] Salvador Trujillo, Don S. Batory, and Oscar Díaz. 2007. Feature Oriented Model Driven Development: A Case Study for Portlets. In *Proc. of ICSE'07*. IEEE Computer Society, Washington, DC, USA, 44–53.

[48] Salvador Trujillo, Jose Miguel Garate, Roberto Erick Lopez-Herrejon, Xabier Mendialdua, Albert Rosado, Alexander Egyed, Charles W. Krueger, and Josune De Sosa. 2010. Coping with Variability in Model-Based Systems Engineering: An Experience in Green Energy. In *Proc. of ECMFA'10 (LNCS)*, Vol. 6138. Springer, 293–304.

[49] Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha S. Gokhale, and Douglas C. Schmidt. 2009. Improving Domain-Specific Language Reuse with Software Product Line Techniques. *IEEE Software* 26, 4 (2009), 47–53.

[50] Xcore. 2018. https://wiki.eclipse.org/Xcore.

[51] Bobbi Young, Judd Cheatwood, Todd Peterson, Rick Flores, and Paul C. Clements. 2017. Product Line Engineering Meets Model Based Engineering in the Defense and Automotive Industries. In *Proc. of SPLC'17*. ACM, 175–179.

[52] Steffen Zschaler, Pablo Sánchez, João Pedro Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. 2009. VML* - A Family of Languages for Variability Management in Software Product Lines. In *Proc. of SLE'09 (LNCS)*, Vol. 5969. Springer, 82–102.