Extensible and modular abstract syntax for feature modeling based on language constructs

Jose-Miguel Horcas
CAOSD group, ITIS, Univ. de Málaga
Andalucía Tech, Spain
horcas@lcc.uma.es

Mónica Pinto
CAOSD group, ITIS, Univ. de Málaga
Andalucía Tech, Spain
pinto@lcc.uma.es

Lidia Fuentes
CAOSD group, ITIS, Univ. de Málaga
Andalucía Tech, Spain
lff@lcc.uma.es

ABSTRACT

Since the definition of feature models in 1990, a large number of language constructs have emerged. Each language construct usually comes with its own abstract and concrete syntax, its semantics, and even its complete language dialect and tool support. Nowadays, there is a consensus in the Software Product Line community about a need for defining a common variability modeling language. But the fact of the matter is that it is very complex to achieve a good compromise between how expressive the language should be and the effort of developing practical tools for a language with all possible language constructs. In this paper, we propose an extensible model-driven engineering approach for defining the abstract syntax of feature modeling language constructs that could be tailored to different needs and domains. We formalize our approach as a set of modular and reusable metamodels that allows practitioners to decide which subset of language constructs to use through: (1) generating a new variability language; and (2) managing feature models with different level of expressiveness. We provide an instantiation and implementation of our approach.

KEYWORDS

Abstract syntax, feature modeling, language construct, language level, metamodeling, model-driven engineering, SPL

1 INTRODUCTION

Feature models stand out as the de-facto standard for specifying variability in *Software Product Lines* (SPLs) [47].

Since the introduction of feature models 30 years ago [38], numerous language constructs have been proposed to increase their expressiveness [13], for example, *cardinality-based variability* [25, 42], *multi-features* [25]; *numerical features* [43] or *non-Boolean features*; *attributed feature models* [14]; and *complex constraints* [4]. Each of those language constructs usually comes with its syntax and semantics, and often coincides with the definition of a completely new modeling language or a new tool to support it [60]. This plethora of extensions, notations, and tools for feature models have brought forward a growing interest in the SPL community about the necessity of defining a single feature modeling language [15]. However, the complexity of defining such a common language resides on reaching a compromise between, (1) its applicability to several domains with different requirements [62], *i.e.,* the expressiveness of its abstract syntax; and (2) the effort of building practical tools that can support all language constructs [36].

To tackle this problem, recent studies [60, 62] are proposing the use of extensible and modular language designs in terms of *language levels* that allow deciding what part of the language we want to keep and use. In domain-specific languages (DSLs) (*e.g.,* MontiCore [21], NIVEL [6]), language constructs are defined in separated models with extension points. Then, a language is defined as the composition of several language constructs. Nevertheless, for feature modeling, there is a lack of actionable solutions applying those design concepts with existing development practices, as concluded in [47]. The question that arises at this point is *how can the principles of extensible DSLs be applied to feature modeling languages?* (**RQ1**). Moreover, *what specific language constructs should be considered in the definition of a feature modeling language?* (**RQ2**). This paper answers **RQ1** by proposing an approach based on model-driven engineering (MDE) and metamodeling [8] that applies the concepts of language levels and modular language design to specifying the abstract syntax of different language constructs for feature modeling. We formalize our approach as a set of reusable and modular metamodels that allows practitioners to decide which language constructs they would like to use through: (1) defining a language for variability modeling; and (2) managing feature models with different levels of expressiveness. To answer **RQ2**, we provide a concrete instantiation of our approach implemented in EMF/Ecore [57] (https://github.com/CAOSD-group/splc2020). We decompose well-known language constructs from existing feature modeling languages in a set of metamodels making explicit the relationships between the language constructs and the metamodels.

The paper is structured as follows. Section 2 motivates our approach by discussing recent work about feature modeling. Section 3 formalizes our approach, while in Section 4 we instantiate and implement it. Finally, Section 5 concludes the paper.

2 RELATED WORK AND MOTIVATION

This section reviews related work about feature modeling, it exposes its limitations and compares it with our approach.

2.1 Recent advances in feature modeling

Since the introduction of feature models in 1990 [38], none successful standard has emerged. Despite some attempts of standardization were proposed such as EMF Feature Model [45], CVL [34], and VEL [46], they did not jell for several reasons (*e.g.,* lack of tool support, monolithic solutions, legal and patent-related issues,...). Also, in 2017, the ISO/IEC 26558:2017 was released as a family of industrial standards for variability modeling, but the SPL community still misses a single common variability modeling language [15].

In the last two years, numerous studies have been published pointing to a definition of such unified language (see Table 1). Most of these works focus on studying the requirements of the intended common language from different viewpoints: language constructs [60], language levels of expressiveness [40, 62], best practices for feature modeling [44], usage scenarios [18], constraints modeling [11], and a repository for feature models exchange [30]. Moreover, several reviews and systematic studies have been published covering language constructs of textual variability modeling languages [28, 60], automated analysis of feature models [31], tool support [9, 36], metrics for analyzing variability [29], feature interaction [55], and evolution of feature models [20, 41]. Table 1 details the most relevant works and explains how we take into account these advances to lay the foundation of our approach.

2.2 Language constructs and language levels

There are lots of language constructs for feature modeling which have been extensively reviewed in [16, 28, 31, 36, 51, 60]. In [51] numerous language constructs are formalized; in [28, 60] a complete review of constructs for textual variability modeling languages is presented; tool support for the language constructs is available in [36]; and their analysis capability can be found in [16, 31].

In order to organize the language constructs and differentiate their expressiveness, researchers are proposing different classifications (see Figure 1). Eichelberger et al. [27] propose a comprehensive classification based on two dimensions: expressiveness and analyzability. Recently, other informal classifications have appeared. For instance, Galindo and Benavides [30] distinguish up to five abstract syntax/model levels based on two groups of relationships: hierarchical relationships and cross-tree constraints. Alférez et al. [4, 5] divide the language constructs of a new variability modeling language (VM) into *basic*, *extended*, and *extra variability*. Thüm et al. [62] define, at least, two major levels for feature modeling notations based on the applicable solvers; and for each major level, they define minor levels driven by requirements of real-world domains, and by constructs of existing languages. They also propose the use of orthogonal levels, which do not influence the expressiveness of the feature model, for example, to achieve modularity and evolution.

As summarized in Figure 1, these classifications and levels allow us to make explicit the trade-offs between the expressiveness and the current tool support and analysis capability of the solvers. However, it is not clear if they are enough since they do not provide enough details to be applied in practice. Moreover, there are disagreements about the nomenclature of the levels and the specific constructs each level should support. In conclusion, *language levels*, as well as *modular language designs*, are considered relevant and good choices to build a future variability modeling language [60, 62], and in this paper, to guide the design of our approach.

Table 1: Recent work in feature modeling.

Study: Raatikainen et al. [47]. **Topic:** SPLs and variability modeling (tertiary study). **Year:** 2019 **Description:** Authors identify a need for actionable solutions for practical applicability with existing development practices, instead of novel solutions in any topic of SPLs and variability modeling. They also argue that feature models stand out clearly as the most popular variability models despite a large number of variability models that exist (*e.g.,* decision models, OVM,...). **Our approach:** We put in practice the main ideas of the state-of-the-art variability modeling to face the problems of defining a common language. We use well-known and consolidated technologies such as feature-orientation and MDE metamodeling (see Section 3). MDE provides several practical advantages such as applying model transformations to support interoperability between feature models notations, straightforward serialization of models, or code generation.
Study: H. ter Beek et al. [60]. **Topic:** Textual variability modeling languages. **Year:** 2019 **Description:** Review of textual variability modeling languages (based on [28]) where the language characteristics are classified in five dimensions: configurable elements, constraints, configuration support, scalability support, and language characteristics. Authors also discuss relevant aspects for the design of a future variability modeling language such as an extensible and modular design. **Our approach:** We use most of those language constructs to instantiate our approach and rely on the proposed dimensions to classify our metamodels (see Section 4.1).
Study: Thüm et al [62]. **Topic:** Language levels for feature modeling notations. **Year:** 2019 **Description:** Authors propose the use of language levels with different expressiveness to handle the problems of defining a common language. Two major levels are concretized based on their capability analysis with solvers, but no detailed definition of minor levels are provided. **Our approach:** Practitioners can easily defined language levels based on their needs (*e.g.,* domain requirements, tool support, analysis capability), by selecting the required constructs.
Study: Nešić et al. [44]. **Topic:** Best practices for feature modeling. **Year:** 2019 **Description:** 34 principles for feature modeling with best practices to be taken into account when defining a modeling methodology or process for feature modeling. **Our approach:** The principles may help (1) language designers to instantiate our approach by deciding the decomposition of language constructs (Section 4.1); and (2) practitioners to use our approach when deciding which constructs to use in their languages and tools.
Study: Horcas et al. [36]. **Topic:** SPLs and variability modeling tools. **Year:** 2019 **Description:** Practical review about tool support for variability modeling and SPLs. It concludes that most of the current tools only support basic constructs, while the support for more complex variability is scarce and with too many limitations (see Figure 1). They also define roadmaps for tool interoperability based on different activities (*e.g.,* modeling, analysis, configuration, derivation,...). **Our approach:** It enables model transformations for feature models (*e.g.,* refactorings [59, 61]) to close the gap between high expressive language constructs and the existing tool support as well as to facilitate the implementation of interoperability roadmaps between different tools [36].
Study: Galindo et al. [31]. **Topic:** Automated analysis of feature models. **Year:** 2019 **Description:** A systematic mapping study to overview the field of automated analysis. They argue that analysis techniques are in general mature enough and we need to find practical applications closer to industry. However, existing efficient solvers (*e.g.,* SAT, #SAT, BDD) [58] only support feature models with a low level of expressiveness (*e.g.,* propositional logic) while solvers capable of analyzing complex variability such as numerical features (*e.g.,* SMT, CSP) [56] offer a poor performance with specific operations such as for counting configurations of feature models (see Figure 1). **Our approach:** It can be used to handle the mismatch that exists between the languages' expressiveness and the capability analysis of the solvers, by defining appropriate model transformations.
Study: Batory [11]. **Topic:** Constraints modeling. **Year:** 2019 **Description:** Discussion about the language that should be used to express constraints in a future variability modeling language. He exposes the problems about using the Object Constraints Language (OCL) standard which is tied to MDE [26] and bets for simplicity and reusing in the definition of the constraints expressions, as in the new A_{OCL} [12] constraint language for MDE. **Our approach:** Language designers can freely decide how to model the constraints as an independent module, without forcing the use of OCL. We advocate for separating constraints based on their expressiveness *i.e.,* propositional logic, first-order logic, arithmetic expressions, etc. (Section 4).
Study: Galindo and Benavides [30]. **Topic:** Repository for feature model exchange. **Year:** 2019 **Description:** Authors discuss up to 12 characteristics that a future feature model repository should have. They also list dependencies with language elements that will affect the development of the repository such as the concrete and abstract syntax of the models and their level of expressiveness. **Our approach:** It can lead the format of the feature models to be used in the repository, storing the models in a generic standard format (*e.g., .xmi*) and downloading them in the desired notation through the use of model-to-text transformations.
Study: Villota et al. [64]. **Topic:** A unified variability modeling language. **Year:** 2019 **Description:** It introduces the High-Level Variability Language (HLVL), a unified variability language that follows an orthogonal approach and serves as an intermediate language for variability. **Our approach:** It keeps clear of the drawbacks regarding the expressiveness trade-offs that a unified language brings forward. Our metamodels, as the HLVL language, can also act as an intermediate language to support interoperability between different notations by using model transformations.
Study: Seidl et al. [54]. **Topic:** SPL of feature modeling notations and constraints. **Year:** 2016 **Description:** Authors define an SPL to generate different variants of feature modeling notations and cross-tree constraint languages. They use a *hyper feature model* [52] to specify the variability of the notations; and implement that variability by defining delta languages [49] for both Ecore metamodels and concrete syntax files using DeltaEcore [53]. **Our approach:** It relies on basic Ecore metamodels which can be directly composed by the relations already defined in the Ecore standard (Section 4.2), without the need of introducing additional variability mechanisms (*e.g.,* delta modeling [49]) or specific generation frameworks (*e.g.,* DeltaEcore [53]). Our approach also allows having multiple abstract syntaxes for the same language construct in separate metamodels, so practitioners can choose the most appropriate definition to their needs. This is prevented when using a hyper feature model where the features are mapped, ideally one-to-one, to the delta modules. In any case, the work of Seidl et al. [54] complements our approach and can be seen as a good starting point for instantiating our approach (Section 4.1) and then incorporating our approach into the solution space of their SPL.

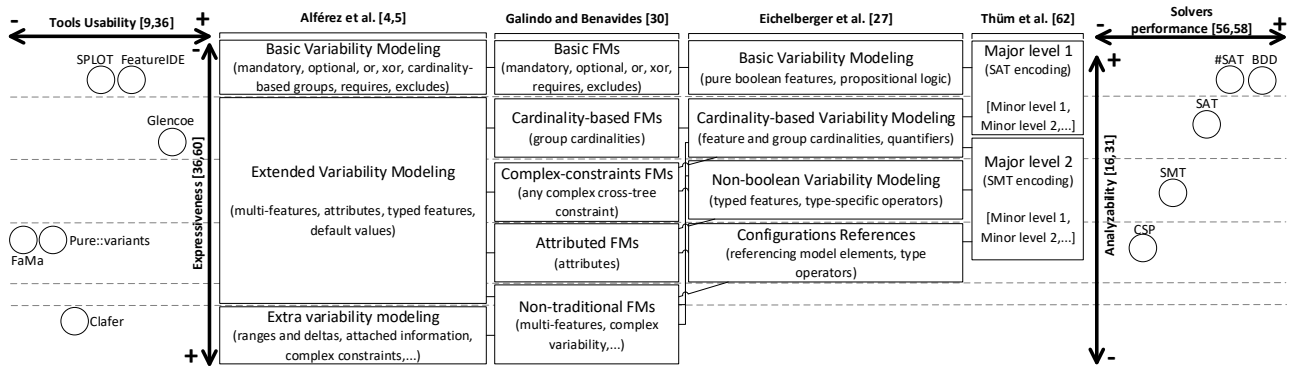


Figure 1: Relations and the trade-off between expressiveness, analyzability, tools support, and solvers, for language constructs.

3 EXTENSIBLE AND MODULAR METAMODELS FOR FEATURE MODELING

We first introduce the concepts used throughout the paper about the formalization of feature models and metamodeling (Section 3.1). Then, we present and formalize our approach (Section 3.2).

3.1 Formal definition of feature models

There exist multiple formalizations for feature models [10, 33, 35, 51]. In this paper, we will use the following definitions:

Definition 3.1 (Feature, feature model, configuration, product, software product line). A *feature* f is a characteristic or end-user-visible behavior of a software system. A *feature model* m is a set of features (F) and their relationships (or dependencies), where a subset $P \subseteq F$ is the set of features that are mapped to artifacts (*i.e., concrete features*). A *configuration* c of a feature model m is a set of its features, *i.e.,* $c \in \mathcal{P}(F)$. A configuration is *valid* if and only if it fulfills all the feature dependencies of m. The set of all valid configurations of m is denoted by C_m. A *product* p is a configuration that contains only concrete features, *i.e.,* $p \in \mathcal{P}(P)$. A *software product line* spl is a set of products, *i.e.,* $spl \in \mathcal{P}(\mathcal{P}(P))$.

Formal languages for feature models. Any modeling language must consist of three elements [33]: a *syntactic domain* ($\mathcal{L}$), a *semantic domain* ($\mathcal{S}$), and a *semantic function* ($\mathcal{M}$). In feature modeling, the *syntactic domain* ($\mathcal{L}$) is the set of all feature models that comply with a given abstract syntax. The *abstract syntax* is a representation of the feature model, which is independent of its physical representation (*i.e.,* the *concrete syntax*). The *semantic domain* ($\mathcal{S}$) specifies the set of all existing product lines, defined as $\mathcal{S} = \mathcal{P}(\mathcal{P}(P))$. The *semantic function* $\mathcal{M} : \mathcal{L} \rightarrow \mathcal{S}$ maps a feature model $m \in \mathcal{L}$ to its software product line $spl \in \mathcal{S}$, denoted by $\mathcal{M}[\![m]\!]$.

Definition 3.2 (Semantics of feature models). The semantics of a feature model m is its set of *valid products*, defined by $[\![m]\!] := \{c \cap P \mid c \in C_m\}$. That is, its software product line.

Expressiveness. The semantic function $\mathcal{M}$ is *total* and is defined for all the elements in $\mathcal{L}$. That means that each feature model in $\mathcal{L}$ represents at least one software product line in $\mathcal{S}$. The inverse function is *partial* and defines the *expressiveness* of the language as the part of the semantic domain that its syntax can express.

Definition 3.3 (Expressiveness). The *expressiveness* of a language $\mathcal{L}$ is the set $E(\mathcal{L}) = \{\mathcal{M}[\![m]\!] \mid m \in \mathcal{L}\}$, also noted $\mathcal{M}[\![\mathcal{L}]\!]$. A language $\mathcal{L}$ with semantic domain $\mathcal{S}$ is *expressively complete* if $E(\mathcal{L}) = \mathcal{S}$, otherwise, $\mathcal{L}$ is *expressively incomplete*. A language $\mathcal{L}_1$ is *more expressive* than a language $\mathcal{L}_2$ if $E(\mathcal{L}_2) \subset E(\mathcal{L}_1)$.

Model-Driven Engineering metamodeling. A *metamodel* [6, 8] specifies the abstract syntax ($\mathcal{L}$) of a modeling language, *e.g.,* the feature modeling language. Therefore, a feature model is an instance of the metamodel used to specify the language $\mathcal{L}$. The semantics of the valid expressions (feature models) produced by the metamodel is given by Definition 3.2.

Definition 3.4 (Well-formed feature model). A feature model m is well-formed (*aka*, correct, well-defined) if m is defined conform to its metamodel and m represents at least one software product line, that may be empty (*i.e., m* is a void feature model). That is, m is an instance of its metamodel respecting all expressions and relationships defined in the metamodel, and $\mathcal{M}[\![m]\!] \neq \emptyset$.

3.2 Formalization of our approach

Our approach consists of the definition of extensible and modular metamodels to describe the abstract syntax of the features models. The language constructs and their relationships are encapsulated in different metamodels with dependencies between them. Given a feature model m we denote M as its corresponding metamodel. The metamodel defines the abstract syntactic domain ($\mathcal{L}$) of the feature models. The set of all feature models that can be specified using the metamodel M is denoted by $\mathcal{L}_M$. That is, the feature model m is a model instance of the metamodel M, denoted by $m \in \mathcal{L}_M$. A metamodel M is defined as a non-empty set of *modeling constructs* (or *language constructs*), *i.e.,* $M = \{l_1, l_2, \ldots, l_t\}$, where each language construct $l_i \in M$ specifies the abstract syntax of a specific variability modeling concept (*e.g.,* optional feature, group feature, requires constraint, multi-feature, attributed feature):

Definition 3.5 (Language construct). A language construct $l \in M$ is the abstract syntax of a specific variability modeling concept.

Examples of language constructs are `Feature` to represent the concept of a feature, `Root` to represent the root feature of the feature model, `OptionalFeature` and `MandatoryFeature` to represent optional and mandatory features, respectively, `AlternativeGroup` for "xor" and `OrGroup` for "or" feature groups, `Multi-Feature` for clonable features [25], `NumericalFeature` for non-Boolean numerical features [43], or `FeatureAttribute` for attributed features [14]. Here, we also define a special language construct for feature models:

Definition 3.6 (Feature model construct). A feature model construct $l_m \in M$ is a language construct that represents the concept of a feature model as a container of other variability modeling concepts such as features and constraints. That is, l_m is the main containment element in the metamodel (*aka*, the root element).[1]

[1] Do not confuse with the `Root` feature language construct of the feature model.

Examples of feature model constructs are `Feature Model` to represent the most generic feature model, `Cardinality-Based FM` to represent feature models with cardinalities [25], `Attributed FM` for models that support features with attributes [14], `Numerical FM` for models with numerical features [43], etc.

Our approach can be seen as a set of modular metamodels that we call $\mathbb{FM}$ and define as follows:

Definition 3.7 ($\mathbb{FM}$). $\mathbb{FM}$ is a set of inter-related metamodels, *i.e.,* $\mathbb{FM} = \{M_0, M_1, M_2, \ldots, M_n\}$, where each metamodel $M_i \in \mathbb{FM}$ is a different n on-empty s et o f l anguage c onstructs, *i.e., M* $_i = \{l_1, l_2, \ldots, l_t\}$ and $M_i \cap M_j = \emptyset, \forall M_i, M_j \in \mathbb{FM}, i \neq j$.

In $\mathbb{FM}$, we define two kinds of relations between language constructs (extension and composition), and a dependency relation between metamodels:

Definition 3.8 (Extension, Composition, D ependency). A language construct $l_i \in M_i$ *extends* another language construct $l_j \in M_j$, noted $l_i <: l_j$, if l_i is a *subtype* of l_j. A language construct $l_i \in M_i$ *is composed* by another language construct $l_j \in M_j$, noted $l_i \models l_j$, if l_i *uses* or *refers* to l_j as part of the definition of l_i. A metamodel $M_i \in \mathbb{FM}$ *depends* on another metamodel $M_j \in \mathbb{FM}$, noted $M_i \Rightarrow M_j$, if $\exists\, l_i \in M_i, l_j \in M_j | \; l_i <: l_j \vee l_i \models l_j$.

The subtype establishes an *is-a* relationship between the language constructs (including multiple inheritance). The composition relation establishes a *usage* or *reference* relationship between the language constructs (including multiple composition). Finally, two metamodels have a dependency between them if there is a construct that extends or uses a construct defined in the other metamodel.

To complete the definition of $\mathbb{FM}$ let us define an initial metamodel $M_0 \in \mathbb{FM}$ with, at least, a feature model construct $l_m \in M_0$ which describes the generic concept of feature model (Definition 3.6).[2] The combination of language constructs, that includes $l_m \in M_0$, allows us to specify well-formed feature models, the semantics of which is defined according to Definition 3.2.

Theorem 3.9. *The language specified by the metamodel $M_0 \in \mathbb{FM}$, noted by $\mathcal{L}_{M_0}$, allows defining, at least, one feature model. That is, $|E(\mathcal{L}_{M_0})| > 0$.*

Proof. M_0 specifies the feature model construct l_m for the concept of the most basic feature model (with no features at all), and therefore, $\mathcal{L}_{M_0}$ defines, at least, the empty product ($\emptyset$). □

Corollary 3.10. *An independent metamodel $M_i \in \mathbb{FM}, i > 0$ is not enough expressive by itself to specify any feature model (*i.e., *$|E(\mathcal{L}_{M_i})| = 0$) unless a construct $l_i \in M_i$ extends directly the feature model construct $l_m \in M_0$, or another construct $l_j \in M_j \mid l_j <: l_m$.*

Corollary 3.11. *In $\mathbb{FM}$, each other metamodel $M_i \in \mathbb{FM}, i > 0$, is related, by the dependency relation ($\Rightarrow$) with, at least, another metamodel $M_j \in \mathbb{FM}$. That is $\forall M_i, i > 0, \exists M_j, j \neq i | M_i \Rightarrow M_j$.*

Theorem 3.9 and Corollaries 3.10 and 3.11 mean that, in order to define well-formed feature models in $\mathbb{FM}$, we need at least an initial metamodel defining a feature model construct l_m (Definition 3.6), and the rest of language constructs can be decomposed in any number of metamodels related somehow with such initial metamodel. However, the use of independent metamodels in isolation cannot define well-formed feature models except they define a feature model construct or extend the feature model construct defined in the initial metamodel (l_m). Because of this, the definition of the different language constructs can be completely modularized in separate but related metamodels.

[2] M_0 may also define any other language construct as shown in Table 2.

4 INSTANTIATING AND IMPLEMENTING $\mathbb{FM}$

4.1 Instantiation of $\mathbb{FM}$

Table 2 presents the instance we propose for $\mathbb{FM}$. First, we consider modeling concepts gathered in the literature (see Section 2.2). Second, we classify the language constructs based on the type of variability they model. For instance, metamodel M_0 contains the FODA [38] concepts for modeling basic variability (excluding constraints). M_2 to M_5 define different types of feature groups. M_6 and M_7 model parent-child relationships. M_8 to M_{10} model non-Boolean feature models [23] such as numerical features [43], attributes [14], or additional information [4]. Third, we put together in the same metamodel those language constructs whose definition directly depend on other constructs (Definition 3.8). However, this is not a strict rule as explained below for the `Group Cardinality` and `Multi-Feature` constructs. Finally, we classify the constructs following four of the dimensions proposed in [60]: configurable elements (metamodels M_0 to M_{14}), constraints support (M_{15} to M_{22}), scalability support (M_{23} to M_{32}), and configurations (M_{33} and M_{34}).

The instance exposes more than 50 language constructs over more than 30 metamodels. Despite the high number of metamodels and language constructs, the modular design of $\mathbb{FM}$ allows practitioners to use just the parts they are interested on. Some metamodels only contain one language construct and it may seem better to group several related language constructs in the same metamodel to reduce the number of metamodels. However, selecting a metamodel implies the inclusion of the abstract syntax of all the language constructs that are part of that metamodel. For this reason, if two language constructs are related among them but there may exist languages that include only one of them it is always better to define them in separated metamodels and then explicitly specify their dependencies. For instance, the `Group Cardinality` and `Multi-Feature` constructs are defined in separate metamodels (M_4 and M_7 respectively) since they are different concepts that languages may not support together. However, both constructs require a `Multiplicity` construct (M_3) to specify a lower and upper bound to define its cardinality. Making explicit the dependencies between the metamodels, as shown in Table 2, allows the automatic selection of the appropriate language constructs to define a specific language with the desired modeling concepts.

4.2 Implementation of $\mathbb{FM}$

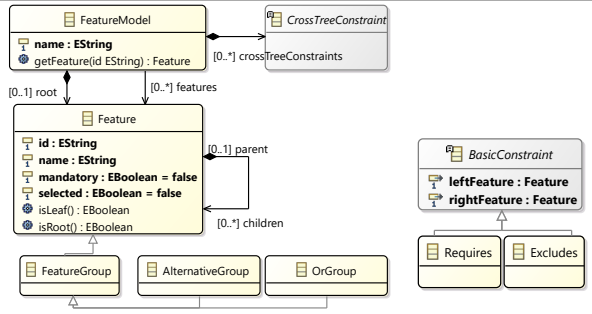
To demonstrate the viability of our approach we propose to implement $\mathbb{FM}$[3] in EMF/Ecore [57] (Figure 2 and Figure 3).

The feature model construct (Definition 3.6) is always defined as a class because it serves as a container of features and constraints. For example, the `FeatureModel` class in the metamodel M_0 specifies the most generic type of feature model, while the `NonBooleanFM` and the `NumericalFM` classes, in M_8 and M_9 respectively (Figure 3), model specializations of the feature model by extending the `FeatureModel` class of M_0. Language constructs (Definition 3.5) can be defined as classes, attributes, or relations. For instance, the `Feature`, `Feature Group`, `Or Group` and `Alternative Group` language constructs of M_0 are defined as classes. But, the

[3] The implementation is available in https://github.com/CAOSD-group/splc2020.

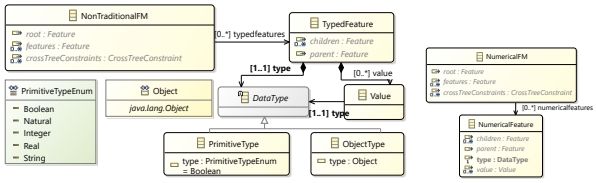
Table 2: An instantiation of $\mathbb{FM}$ with its metamodels (M) classified according to the dimensions (D) proposed in [60], the set of language constructs (modeling concepts), their relationships and dependencies, and examples of languages including them.

D	M	Language constructs	Extends	Dependencies	Description	Examples
Configurable elements	M_0	*Feature Model*			The top element of the metamodel.	FODA [38], and basically almost all variability modeling languages and tools support the constructs of M_0.
		Feature			The unit of variability. A feature can be *optional* or *mandatory*.	
		Root	Feature		The root feature $r \in F$ of the feature model. r is always mandatory.	
		Optional Feature	Feature		It represents an *optional* feature.	
		Mandatory Feature	Feature		It represents a *mandatory* feature.	
		Parent-Child Relationship			Features decomposition. A child can be selected only when its parent is selected.	
		Feature Group	Feature		Children of a feature $f \in F$ can be grouped.	
		Alternative Group	Feature Group		It defines a one-out-of-many choice, *i.e.,* an *xor group* <1 . . 1>.	
		Or Group	Feature Group		It defines a some-out-of-many choice, *i.e.,* an *or group* <1 . . *>.	
		Cross-Tree Constraint			The generic concept of a constraint.	
	M_1	*Abstract Feature*	Feature	M_0	Distinction between concrete and abstract features in leaf features.	Relaxed FMs [40].
	M_2	*Mutex-Group*	Feature Group	M_0	Feature groups where at most one feature can be selected.	KConfig [19], CDL [19].
	M_3	*Cardinality-Based FM*	Feature Model	M_0	It allows defining cardinalities for features and groups.	Cardinality-based FMs [25].
		Multiplicity			It defines lower and upper bounds.	
	M_4	*Group Cardinality*	Feature Group	M_0, M_3	Arbitrary multiplicities <n. .m> for group features, bounded and unbounded (*).	Cardinality-based FMs [25].
	M_5	*Multiple Decomposition Type*	Parent-Child Relationship	M_0	Different group features (*e.g.,* or and xor), below the same feature.	Generative Programming [24].
	M_6	*Directed Acyclic Graph*	Parent-Child Relationship	M_0	Features with multiple parents.	FORM [39], FeatuRSEB [32].
	M_7	*Multi-Feature*	Feature	M_0, M_3	Features with cardinalities (*aka*, clonable features).	Cardinality-based FMs [25].
	M_8	*Non-Boolean FM*	Feature Model	M_0	Definition of arbitrary data types for features and/or attributes.	VM [4], Clafer [37], CVL [34], PyFML [3].
		Data Type			Primitive (*e.g.,* Boolean, Integer, Float, String,. . .), and user-defined types.	
		Value Assignment			It allows providing a value to a specific data type.	
		Typed Feature	Feature		Arbitrary data types for features.	
		Attached Information			Additional information (e.g., attributes, meta-attributes) for configurable elements.	
	M_9	*Numerical FM*	Non-Boolean FM	M_0, M_8	Feature model with numerical features.	Numerical FMs [43].
		Numerical Feature	Typed Feature		Non-Boolean numerical features (*e.g.,* Natural, Integer, Real,. . .).	
	M_{10}	*Attributed FM*	Non-Boolean FM	M_0, M_8	Feature models with attributes.	TVL [22], FaMa [17].
		Feature Attribute	Attached Information		Features with attributes (*e.g.,* cost, performance).	
	M_{11}	*Binding time*	Attached Information	M_0, M_8	Point time when the variability decision must be made.	IVML [50].
	M_{12}	*Default Value*	Value Assignment	M_0, M_8	It allows establishing a default value to a typed feature or attribute.	PyFML [3].
	M_{13}	*Delta Value*	Value Assignment	M_0, M_8, M_9, M_{18}	It reduces the number of acceptable numeric values.	VM [4].
	M_{14}	*Range*	Value Assignment	M_0, M_8, M_9, M_{18}	It allows defining ranges of values for numerical features or attributes.	VM [4].
Constraints	M_{15}	*Simple dependency*	Cross-Tree Constraint	M_0	*Requires* and *excludes* constraints.	FODA [38], FDL [63].
	M_{16}	*Propositional logic constraint*	Cross-Tree Constraint	M_0	Arbitrary propositional formulas over the features ($\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$).	Almost all languages.
	M_{17}	*First-order logic constraint*	Cross-Tree Constraint	M_0, M_3, M_7	Quantifiers ($\forall, \exists$), predicates, functions, and constants.	Forfamel [7], Clafer [37].
	M_{18}	*Relational expressions*	Cross-Tree Constraint	M_0, M_8	Operators for comparing features or attributes (==, <, <=, >, >=, $\neq$, !).	Clafer [37], PyFML [3].
	M_{19}	*Arithmetic expressions*	Cross-Tree Constraint	M_0, M_8	Arithmetic formulas, functions, and operators (+, −, ×, /, %,. . .).	Clafer [37], PyFML [3].
	M_{20}	*Cardinalities expressions*	Cross-Tree Constraint	M_0, M_3, M_4	Cardinalities expressed in terms of constraints.	Clafer [37].
	M_{21}	*Type restrictions*	Cross-Tree Constraint	M_0, M_8	Type-specific operators for constraints (*e.g.,* String operators, regular expressions).	CVL (OCL) [34].
	M_{22}	*Default constraints*	Cross-Tree Constraint	M_0	Constraints that can be altered as part of the constraint-resolution process.	IVML [50].
Scalability	M_{23}	*Compositional FM*	Feature Model	M_0, M_8	Mechanisms for composition and inheritance for large feature models.	Clafer [37], FAMILIAR [2], CVL [34], VELVET [48].
		Interface			The concept of interface of feature model for modularization.	
		Scope			Support for declaring scopes (*e.g.,* scoped import of models).	
	M_{24}	*Configuration Reference*	Interface	M_0, M_{23}	It defines links between models and configurable elements.	CVL [34], VSL [1].
	M_{25}	*Containment Feature*	Interface	M_0, M_{23}	Composition on type level (*aka*, composite units).	CVL [34], Clafer [37].
	M_{26}	*Imports*		M_0, M_{23}	Import of models.	IVML [50], VM [4].
	M_{27}	*Merge*		M_0, M_{23}	Operator for overlapping.	FAMILIAR [2].
	M_{28}	*Aggregate*		M_0, M_{23}	Operator for disjoint models.	FAMILIAR [2].
	M_{29}	*Include*		M_0, M_{23}	Models can be composed of a model of a larger scale.	PyFML [3], TVL [22].
	M_{30}	*Model Version*	Non-Boolean FM	M_0, M_8	Support for managing versions of the models.	IVML [50].
	M_{31}	*Visibility*	Non-Boolean FM	M_0, M_8	Visibility (*e.g.,* public, private) for configurable elements.	VSL [1].
	M_{32}	*View-Points*	Non-Boolean FM	M_0, M_8	Multiple view-points for feature models.	VELVET [48].
C.	M_{33}	*FM Configuration*		M_0	A full configuration of a feature model as a selection (and assignment) of features.	Almost all languages.
	M_{34}	*FM Partial Configuration*	Configuration	M_0, M_{33}	A partial configuration of a feature model.	Clafer [37], VM [4].



(a) Metamodel M_0. **(b) Metamodel M_{15}.**

Figure 2: $\mathbb{FM}$ metamodels for Basic Feature Models.



(a) Metamodel M_8. **(b) Metamodel M_9.**

Figure 3: $\mathbb{FM}$ metamodels for Numerical Feature Models.

`Optional Feature` and `Mandatory Feature` language constructs are implemented as a unique attribute of the `Feature` class, while the `Root` and `Parent-Child Relationship` constructs are defined as composition references. Finally, the `FeatureModel` class in M_0 exposes the `CrossTreeConstraint` abstract class using the composition relation that enables specifying constraints in separate metamodels as in M_{15} where `BasicConstraint` extends it, and `Requires` and `Excludes` extend `BasicConstraint`, all of them using the *Super Type* relation. The same design is used in M_8 and M_9 to define generic concepts for non-Boolean features (M_8), and specializing them for numerical features (M_9).

5 CONCLUSIONS AND ONGOING WORK

Our approach advocates for a modular and extensible design that allows practitioners to decide which language levels support based on their needs. Using just the needed language constructs may lead to better performance and analysis capabilities that are not available for a full language with all possible constructs.

Our ongoing work includes taking advantages of MDE techniques (*e.g.,* model transformations) at the language construct level to support and automatize activities such as the interoperability between existing languages, tools, and solvers with different level of expressiveness and analyzability, language analysis (*e.g.,* expressiveness, succinctness, embeddability), the evolution of feature models (*e.g.,* edits and refactorings), and benchmarking (*e.g.,* automated tests). We also plan to evaluate the applicability and usefulness of our approach by covering the usage scenarios for a common feature modeling language identified in the SPL community [18].

ACKNOWLEDGMENTS

Work supported by *Magic* P12-TIC1814, *MEDEA* RTI2018-099213-B-I00 (co-financed by FEDER), *Rhea* P18-FR-1081 (MCI/AEI/FEDER, UE), *LEIA* UMA18-FEDERIA-157, and *TASOVA* MCIU-AEI TIN2017-90644-REDT.

REFERENCES

[1] Andreas Abele, Yiannis Papadopoulos, David Servat, Martin Törngren, and Matthias Weber. 2010. The CVM Framework - A Prototype Tool for Compositional Variability Management. In *Fourth International Workshop on Variability Modelling of Software-Intensive Systems (ICB-Research Report)*, Vol. 37. Universität Duisburg-Essen, 101–105. http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. FAMILIAR: A domain-specific language for large scale management of feature models. *Sci. Comput. Program.* 78, 6 (2013), 657–681. https://doi.org/10.1016/j.scico.2012.12.004

[3] A. F. Al-Azzawi. 2018. PyFml - a Textual Language For Feature Modeling. *International Journal of Software Engineering Applications (IJSEA)* 9, 1 (2018), 41–53. https://doi.org/10.5121/ijsea.2018.9104

[4] Mauricio Alférez, Mathieu Acher, José A. Galindo, Benoit Baudry, and David Benavides. 2019. Modeling variability in the video domain: language and experience report. *Software Quality Journal* 27, 1 (2019), 307–347. https://doi.org/10.1007/s11219-017-9400-8

[5] Mauricio Alférez, José Angel Galindo Duarte, Mathieu Acher, and Benoit Baudry. 2014. *Modeling Variability in the Video Domain: Language and Experience Report.* Research Report RR-8576. INRIA. https://hal.inria.fr/hal-01023159

[6] Timo Asikainen and Tomi Männistö. 2009. Nivel: a metamodelling language with a formal semantics. *Software and Systems Modeling* 8, 4 (2009), 521–549. https://doi.org/10.1007/s10270-008-0103-2

[7] Timo Asikainen, Tomi Männistö, and Timo Soininen. 2006. A Unified Conceptual Foundation for Feature Modelling. In *10th International Software Product Line Conference (SPLC 2006)*. IEEE Computer Society, 31–40. https://doi.org/10.1109/SPLINE.2006.1691575

[8] C. Atkinson and T. Kuhne. 2003. Model-driven development: a metamodeling foundation. *IEEE Software* 20, 5 (2003), 36–41.

[9] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher, and Goetz Botterweck. 2017. CASE Tool Support for Variability Management in Software Product Lines. *ACM Comput. Surv.* 50, 1 (2017), 14:1–14:45. https://doi.org/10.1145/3034827

[10] Don S. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *9th International Software Product Line Conference (SPLC 2005) (Lecture Notes in Computer Science)*, Vol. 3714. Springer, 7–20. https://doi.org/10.1007/11554844_3

[11] Don S. Batory. 2019. Should future variability modeling languages express constraints in OCL?. In *23rd International Systems and Software Product Line Conference (SPLC 2019), Volume B*, Carlos Cetina, Oscar Díaz, Laurence Duchien, Marianne Huchard, Rick Rabiser, Camille Salinesi, Christoph Seidl, Xhevahire Tërnava, Leopoldo Teixeira, Thomas Thüm, and Tewfik Ziadi (Eds.). ACM, 87:1. https://doi.org/10.1145/3307630.3342406

[12] Don S. Batory and Najd Altoyan. 2020. Aocl : A Pure-Java Constraint and Transformation Language for MDE. In *8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2020)*. SCITEPRESS, 319–327. https://doi.org/10.5220/0008942803190327

[13] David Benavides. 2019. Variability Modelling and Analysis During 30 Years. In *From Software Engineering to Formal Methods and Tools, and Back (Lecture Notes in Computer Science)*, Vol. 11865. Springer, 365–373. https://doi.org/10.1007/978-3-030-30985-5_21

[14] David Benavides, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés. 2005. Automated Reasoning on Feature Models. In *17th International Conference on Advanced Information Systems Engineering (CAiSE 2005) (Lecture Notes in Computer Science)*, Vol. 3520. Springer, 491–503. https://doi.org/10.1007/11431855_34

[15] David Benavides, Rick Rabiser, Don S. Batory, and Mathieu Acher. 2019. First international workshop on languages for modelling variability (MODEVAR 2019). In *23rd International Systems and Software Product Line Conference (SPLC 2019), Volume A*. ACM, 46:1. https://doi.org/10.1145/3336294.3342364

[16] David Benavides, Sergio Segura, and Antonio Ruiz Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.* 35, 6 (2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001

[17] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz Cortés. 2007. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *First International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2007)*. 129–134.

[18] Thorsten Berger and Philippe Collet. 2019. Usage scenarios for a common feature modeling language. In *23rd International Systems and Software Product Line Conference (SPLC 2019), Volume B*. ACM, 86:1–86:8. https://doi.org/10.1145/3307630.3342403

[19] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. Software Eng.* 39, 12 (2013), 1611–1640. https://doi.org/10.1109/TSE.2013.34

[20] Vinícius Bischoff, Kleinner Farias, Lucian José Gonçales, and Jorge Luis Victória Barbosa. 2019. Integration of feature models: A systematic mapping study. *Inf. Softw. Technol.* 105 (2019), 209–225. https://doi.org/10.1016/j.infsof.2018.08.016

[21] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2018. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *12th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2018)*, Rafael Capilla, Malte Lochau, and Lidia Fuentes (Eds.). ACM, 75–82. https://doi.org/10.1145/3168365.3168368

[22] Andreas Classen, Quentin Boucher, and Patrick Heymans. 2011. A text-based approach to feature modelling: Syntax and semantics of TVL. *Sci. Comput. Program.* 76, 12 (2011), 1130–1143. https://doi.org/10.1016/j.scico.2010.10.005

[23] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2013. Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 472–481. https://doi.org/10.1109/ICSE.2013.6606593

[24] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative programming - methods, tools and applications*. Addison-Wesley. http://www.addison-wesley.de/main/main.asp?page=englisch/bookdetails&productid=99258

[25] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. 2005. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice* 10, 1 (2005), 7–29. https://doi.org/10.1002/spip.213

[26] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. 2006. Feature Models are Views on Ontologies. In *10th International Software Product Line Conference (SPLC 2006)*. IEEE Computer Society, 41–51. https://doi.org/10.1109/SPLINE.2006.1691576

[27] Holger Eichelberger, Christian Kröher, and Klaus Schmid. 2013. An Analysis of Variability Modeling Concepts: Expressiveness vs. Analyzability. In *13th International Conference on Software Reuse (ICSR 2013) (Lecture Notes in Computer Science)*, Vol. 7925. Springer, 32–48. https://doi.org/10.1007/978-3-642-38977-1_3

[28] Holger Eichelberger and Klaus Schmid. 2015. Mapping the design-space of textual variability modeling languages: a refined analysis. *STTT* 17, 5 (2015), 559–584. https://doi.org/10.1007/s10009-014-0362-x

[29] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. 2019. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Inf. Softw. Technol.* 106 (2019), 1–30. https://doi.org/10.1016/j.infsof.2018.08.015

[30] José A. Galindo and David Benavides. 2019. Towards a new repository for feature model exchange. In *23rd International Systems and Software Product Line Conference (SPLC 2019), Volume B*. ACM, 85:1–85:4. https://doi.org/10.1145/3307630.3342405

[31] José A. Galindo, David Benavides, Pablo Trinidad, Antonio Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated analysis of feature models: Quo vadis? *Computing* 101, 5 (2019), 387–433. https://doi.org/10.1007/s00607-018-0646-1

[32] Martin L. Griss, John M. Favaro, and Massimo D'Alessandro. 1998. Integrating feature modeling with the RSEB. In *5th International Conference on Software Reuse (ICSR 1998)*. IEEE Computer Society, 76–85. https://doi.org/10.1109/ICSR.1998.685732

[33] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer* 37, 10 (2004), 64–72. https://doi.org/10.1109/MC.2004.172

[34] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. 2008. Adding Standardized Variability to Domain Specific Languages. In *12th International Software Product Line Conference (SPLC 2008)*. IEEE Computer Society, 139–148. https://doi.org/10.1109/SPLC.2008.25

[35] Patrick Heymans, Pierre-Yves Schobbens, Jean-Christophe Trigaux, Raimundas Matulevicius, Andreas Classen, and Yves Bontemps. 2007. Towards the comparative evaluation of feature diagram languages. In *Software and Services Variability Management Workshop Concepts, Models and Tools (SVM 2007)*.

[36] José-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2019. Software product line engineering: a practical experience. In *23rd International Systems and Software Product Line Conference (SPLC 2019), Volume A*. ACM, 25:1–25:13. https://doi.org/10.1145/3336294.3336304

[37] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2019. Clafer: Lightweight Modeling of Structure, Behaviour, and Variability. *Programming Journal* 3, 1 (2019), 2. https://doi.org/10.22152/programming-journal.org/2019/3/2

[38] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst. Technical Report CMU/SEI-90-TR-21.

[39] Kyo Chul Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Ann. Software Eng.* 5 (1998), 143–168. https://doi.org/10.1023/A:1018980625587

[40] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is there a mismatch between real-world feature models and

product-line research?. In *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, 291–302. https://doi.org/10.1145/3106237.3106252

[41] Maira Marques, Jocelyn Simmonds, Pedro O. Rossel, and María Cecilia Bastarrica. 2019. Software product line evolution: A systematic literature review. *Inf. Softw. Technol.* 105 (2019), 190–208. https://doi.org/10.1016/j.infsof.2018.08.014

[42] Raphaël Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. 2011. A formal semantics for feature cardinalities in feature diagrams. In *Fifth International Workshop on Variability Modelling of Software-Intensive Systems, Namur, Belgium, January 27-29, 2011. Proceedings (ACM International Conference Proceedings Series)*, Patrick Heymans, Krzysztof Czarnecki, and Ulrich W. Eisenecker (Eds.). ACM, 82–89. https://doi.org/10.1145/1944892.1944902

[43] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don S. Batory. 2019. Uniform random sampling product configurations of feature models that have numerical features. In *23rd International Systems and Software Product Line Conference (SPLC 2019), Volume A*. ACM, 39:1–39:13. https://doi.org/10.1145/3336294.3336297

[44] Damir Nesic, Jacob Krüger, Stefan Stanciulescu, and Thorsten Berger. 2019. Principles of feature modeling. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE 2019)*. ACM, 62–73. https://doi.org/10.1145/3338906.3338974

[45] Holger Papajewski, Benjamin Klatt, Andreas Graf, and André Maaß. 2014. EMF Feature Model. https://projects.eclipse.org/projects/modeling.emft.featuremodel.

[46] SPES_XT project. 2015. The Variability Exchange Language (VEL). https://www.variability-exchange-language.org/.

[47] Mikko Raatikainen, Juha Tiihonen, and Tomi Männistö. 2019. Software product lines and variability modeling: A tertiary study. *J. Syst. Softw.* 149 (2019), 485–510. https://doi.org/10.1016/j.jss.2018.12.027

[48] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. 2011. Multi-dimensional variability modeling. In *5th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2011) (ACM International Conference Proceedings Series)*. ACM, 11–20. https://doi.org/10.1145/1944892.1944894

[49] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings (Lecture Notes in Computer Science)*, Jan Bosch and Jaejoon Lee (Eds.), Vol. 6287. Springer, 77–91. https://doi.org/10.1007/978-3-642-15579-6_6

[50] Klaus Schmid, Christian Kröher, and Sascha El-Sharkawy. 2018. Variability modeling with the integrated variability modeling language (IVML) and EASyproducer. In *22nd International Systems and Software Product Line Conference (SPLC 2018), Volume 1*. ACM, 306. https://doi.org/10.1145/3233027.3233057

[51] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic semantics of feature diagrams. *Comput. Networks* 51, 2 (2007), 456–479. https://doi.org/10.1016/j.comnet.2006.08.008

[52] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Capturing variability in space and time with hyper feature models. In *The Eighth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '14, Sophia Antipolis, France, January 22-24, 2014*, Philippe Collet, Andrzej Wasowski, and Thorsten Weyer (Eds.). ACM, 6:1–6:8. https://doi.org/10.1145/2556624.2556625

[53] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. DeltaEcore - A Model-Based Delta Language Generation Framework. In *Modellierung 2014, 19.-21. März 2014, Wien, Österreich (LNI)*, Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer (Eds.), Vol. P-225. GI, 81–96. https://dl.gi.de/20.500.12116/17067

[54] Christoph Seidl, Tim Winkelmann, and Ina Schaefer. 2016. A Software Product Line of Feature Modeling Notations and Cross-Tree Constraint Languages. In *Modellierung 2016, 2.-4. März 2016, Karlsruhe (LNI)*, Andreas Oberweis and Ralf H. Reussner (Eds.), Vol. P-254. GI, 157–172. https://dl.gi.de/20.500.12116/821

[55] Larissa Rocha Soares, Pierre-Yves Schobbens, Ivan do Carmo Machado, and Eduardo Santana de Almeida. 2018. Feature interaction in software product line engineering: A systematic mapping study. *Inf. Softw. Technol.* 98 (2018), 44–58. https://doi.org/10.1016/j.infsof.2018.01.016

[56] Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, and Ina Schaefer. 2020. SMT-based variability analyses in FeatureIDE. In *14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2020)*. ACM, 6:1–6:9. https://doi.org/10.1145/3377024.3377036

[57] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

[58] Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2020. Evaluating #SAT solvers on industrial feature models. In *14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2020)*, Maxime Cordy, Mathieu Acher, Danilo Beuche, and Gunter Saake (Eds.). ACM, 3:1–3:9. https://doi.org/10.1145/3377024.3377025

[59] Mohammad Tanhaei, Jafar Habibi, and Seyed-Hassan Mirian-Hosseinabadi. 2016. Automating feature model refactoring: A Model transformation approach. *Inf. Softw. Technol.* 80 (2016), 138–157. https://doi.org/10.1016/j.infsof.2016.08.011

[60] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. 2019. Textual variability modeling languages: an overview and considerations. In *23rd International Systems and Software Product Line Conference (SPLC 2019), Volume B*. ACM, 82:1–82:7. https://doi.org/10.1145/3307630.3342398

[61] Thomas Thüm, Don S. Batory, and Christian Kästner. 2009. Reasoning about edits to feature models. In *31st International Conference on Software Engineering (ICSE 2009)*. IEEE, 254–264. https://doi.org/10.1109/ICSE.2009.5070526

[62] Thomas Thüm, Christoph Seidl, and Ina Schaefer. 2019. On language levels for feature modeling notations. In *23rd International Systems and Software Product Line Conference (SPLC 2019), Volume B*. ACM, 83:1–83:4. https://doi.org/10.1145/3307630.3342404

[63] Arie Van Deursen and Paul Klint. 2002. Domain-specific language design requires feature descriptions. *Journal of computing and information technology* 10, 1 (2002), 1–17. https://doi.org/10.2498/cit.2002.01.01

[64] Ángela Villota, Raúl Mazo, and Camille Salinesi. 2019. The high-level variability language: an ontological approach. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume B, Paris, France, September 9-13, 2019*, Carlos Cetina, Oscar Díaz, Laurence Duchien, Marianne Huchard, Rick Rabiser, Camille Salinesi, Christoph Seidl, Xhevahire Tërnava, Leopoldo Teixeira, Thomas Thüm, and Tewfik Ziadi (Eds.). ACM, 84:1–84:8. https://doi.org/10.1145/3307630.3342401