The Journal of Systems & Software 197 (2023) 111579

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A modular metamodel and refactoring rules to achieve software product line interoperability☆,☆☆

Jose-Miguel Horcas *, Mónica Pinto, Lidia Fuentes

Andalucía Tech, ITIS Software, Universidad de Málaga, Spain

ARTICLE INFO

Article history:
Received 13 January 2022
Received in revised form 27 September 2022
Accepted 30 November 2022
Available online 5 December 2022

Dataset link: https://github.com/CAOSD-group/rhea

Keywords:
Variability modeling language
Modular metamodel
Model refactoring
Model specialization
Interoperability
Edge computing

ABSTRACT

Emergent application domains, such as cyber–physical systems, edge computing or industry 4.0. present a high variability in software and hardware infrastructures. However, no single variability modeling language supports all language extensions required by these application domains (*i.e.,* attributes, group cardinalities, clonables, complex constraints). This limitation is an open challenge that should be tackled by the software engineering field, and specifically by the software product line (SPL) community. A possible solution could be to define a completely new language, but this has a high cost in terms of adoption time and development of new tools. A more viable alternative is the definition of refactoring and specialization rules that allow interoperability between existing variability languages. However, with this approach, these rules cannot be reused across languages because each language uses a different set of modeling concepts and a different concrete syntax. Our approach relies on a modular and extensible metamodel that defines a common abstract syntax for existing variability modeling extensions. We map existing feature modeling languages in the SPL community to our common abstract syntax. Using our abstract syntax, we define refactoring rules at the language construct level that help to achieve interoperability between variability modeling languages.

1. Introduction

Emergent application domains, such as cyber–physical systems (Fadhlillah et al., 2021; Krüger et al., 2017; Meixner et al., 2019), edge computing (Cañete et al., 2020; Liu et al., 2019), or industry 4.0 (Wortmann et al., 2020) present a high variability in terms of software infrastructures (e.g., different kinds of virtual machines or platforms) and hardware (e.g., IoT devices, cloud servers or edge devices). Specifying such diversity in formal models allows us to manage variability across products in a software product line (SPL). There are a large number of formal models to specify variability such as *feature models* (Kang et al., 1990), *decision models* (Schmid et al., 2011), or *OVM* (Pohl et al., 2005), but feature models stand out as the most popular ones and can be considered as the de-facto standard for specifying variability in SPLs (Raatikainen et al., 2019). Thus, in this paper, we focus on feature models and feature modeling languages. *Feature models* allow to formally specify the variability of a system in terms of a set of features and the relationships and constraints among them (Alférez et al., 2019). However, the domains mentioned above require advanced feature modeling concepts (or *language constructs*) to fulfill their specific requirements, beyond those defined for *basic variability modeling* (Alférez et al., 2019) — *i.e.,* *optional* and *mandatory* features to decide whether a feature must always be present in a product or not, *alternative* (“xor”) and *selection* (“or”) groups to decide exclusive feature selections and allowed combinations of features in a product, and *requires* or *excludes* constraints to specify whether a feature must be present or excluded in the presence of another feature.

The necessity of using advanced variability modeling can be easily identified in the edge computing domain. Edge computing brings computation from data centers towards the edge of the network, exploiting smart objects, mobile phones or network gateways to perform application tasks, providing services on behalf of the cloud (Merenda et al., 2020). Edge computing presents variability at different levels, and this variability needs to be modeled using advanced variability constructs. For instance, the functionality level of an edge computing application is usually defined in terms of a set of tasks (*e.g.,* data capture, processing, mixing, filtering) where the data type, the data capturing frequency, or the intermediate buffers’ size present

☆ Work supported by the projects MEDEA RTI2018-099213-B-I00, IRIS PID2021-122812OB-I00 (co-financed by FEDER funds), Rhea P18-FR-1081 (MCI/AEI/FEDER, UE), LEIA UMA18-FEDERIA-157, and DAEMON H2020-101017109. We would also like to thank Miguel de la Morena Pérez who worked on the implementation of this study as part of the Rhea project and his Master Dissertation. Funding for open access: Universidad de Málaga / CBUA.

☆☆ Editor: Heiko Koziolek.

* Corresponding author.
E-mail addresses: horcas@lcc.uma.es (J.-M. Horcas), pinto@lcc.uma.es (M. Pinto), lff@lcc.uma.es (L. Fuentes).

https://doi.org/10.1016/j.jss.2022.111579

multiple variants in terms of non-Boolean features (*e.g.,* the *numerical feature* (Munoz et al., 2019) advanced construct). Also, the hardware level, where the edge computing application is deployed, requires to configure multiple similar devices with different configurations (*i.e.,* it requires the use of *features with cardinalities* or *multi-features* (Czarnecki et al., 2005), which is also an advanced construct). The same occurs with the network or the software infrastructure levels. All this variability needs to be formally represented in order to be able to reason about it (*e.g.,* counting valid products, validating partial configurations,…) and find the best configuration according to certain criteria (*e.g.,* energy efficiency). Moreover, once a particular configuration is created from the variability model, the code to be included in each part of the system should be automatically generated from the code of each selected feature. Thus, the language needs to provide a mapping between the variability model and the artifacts that implement the features.

Unfortunately, current variability modeling languages and tools support these advanced characteristics only partially, as demonstrated in Horcas et al. (2022). For instance, for the scenario described above, several tools can be used to create the variability model, such as FeatureIDE (Meinicke et al., 2017), Glencoe (Schmitt et al., 2018), UVL (Sundermann et al., 2021), FaMa (Benavides et al., 2007), or Clafer (Juodisius et al., 2019), but only Clafer provides support for modeling advanced characteristics such as numerical features and multi-features. On the other hand, languages and tools with high expressiveness such as Clafer (Juodisius et al., 2019) offer poor performance when reasoning about variability in large configuration spaces (*e.g.,* counting configurations for large industrial models). While tools with good support for automatic reasoning, such as Glencoe (Schmitt et al., 2018) or FaMa (Benavides et al., 2007) are less expressive, only supporting basic variability constructs (Munoz et al., 2021). In addition, from the above tools, only FeatureIDE (Meinicke et al., 2017) can be used to automatically generate the application code for a particular configuration. To sum up, none of them provide full tool support to model, reason, and implement variability for modern application domains, such as edge computing, our motivating case study.

In this paper, we propose an approach to address the aforementioned problem based on model-driven engineering (MDE) and metamodeling (Atkinson and Kuhne, 2003) and guided by the language constructs of feature modeling languages. Despite the definition of a new feature modeling language (Benavides et al., 2019; Sundermann et al., 2021) that incorporates all language constructs may express all the variability required for most existing domains, it requires too much effort to develop tools supporting all its characteristics. In addition, the resulting feature models could be very difficult to formalize and analyze with existing solvers. On the opposite side, a succinct core language with basic constructs, easy to formalize, and that can be supported by many tools is most likely not sufficient to express the requirements of industrial domains, as demonstrated in Knüppel et al. (2017) and illustrated in the edge computing domain (see Section 2). *Our approach provides interoperability between existing feature modeling languages. It relies on the idea of having a common, general, and modular abstract syntax (Horcas et al., 2020; Schobbens et al., 2007) of existing language constructs to model variability. Interoperability is achieved by representing existing feature modeling languages in the common abstract syntax and providing model transformations at the language construct level in terms of the abstract syntax (see Section 3).*

Since the introduction of feature models in 1990 (Kang et al., 1990), the number of language constructs has increased considerably to cope with the requirements of many different application domains (Horcas et al., 2022; Galster et al., 2014), including the edge computing domain. Each of these language constructs (*e.g., cardinality-based variability* (Czarnecki et al., 2005), *multi-features* (Czarnecki et al., 2005); *numerical features* (Munoz et al., 2019) or *non-Boolean features*; *attributed feature models* (Benavides et al., 2005); and *complex constraints* (Alférez et al., 2019)) usually comes with its own syntax and semantics, and often coincides with the definition of a completely new modeling language or a new tool to support it (ter Beek et al., 2019). The complexity of defining a common language is caused by reaching a compromise between, (1) its applicability to several domains with different requirements (Thüm et al., 2019), *i.e.,* the expressiveness of its abstract syntax; and (2) the effort to develop practical tools that can support all language constructs (Horcas et al., 2022). *We present the theory of extensible and modular language constructs for feature modeling defined in Horcas et al. (2020) and use it to define a common abstract syntax with a set of well-known language constructs in the SPL community (ter Beek et al., 2019; Horcas et al., 2020). Then we apply it to represent the existing feature modeling languages in the proposed common abstract syntax (see Section 4).*

Interoperability and translations between feature modeling languages are achieved by the definition of edits (Thüm et al., 2009) and model transformations (Feichtinger, 2021; Feichtinger and Rabiser, 2020). However, transformations between language constructs are not always possible without losing information (*e.g.,* a numerical feature may be translated into a Boolean feature by discretizing its values in intervals). Although some *refactorings* (*i.e.,* transformations without lose of information) have been studied in the literature (Knüppel et al., 2017), they are based on the concrete syntax of a specific feature model language and, furthermore, have been formalized only in theory without demonstrating its viability in practice. *As a proof of concept, we demonstrate the feasibility and applicability of our proposal by providing (1) an implementation of the common abstract syntax as a set of modular metamodels (Atkinson and Kuhne, 2003) using the EMF/Ecore framework (Steinberg et al., 2008); and (2) two different implementations of a subset of the existing refactorings between language constructs to provide interoperability between feature models (see Section 6).*

The rest of the paper is organized as follows. Section 2 presents the state of the art and motivates our approach with a case study to better illustrate the requirements that emergent systems impose in the definition of feature modeling languages. Then Section 3 describes our approach based on a common abstract syntax (called $\mathbb{CAF}$) to achieve interoperability. Section 4 formalizes $\mathbb{CAF}$, illustrates it with existing language constructs, and map well-known feature modeling dialects to our common abstract syntax. Section 5 describes the model transformations between language constructs focusing on the refactorings that have been formalized in the literature. Section 6 provides an implementation of $\mathbb{CAF}$ as a proof of concept and evaluates our approach. Finally, Section 7 concludes the paper.

2. State-of-the-art and motivation

As previously stated, the motivation of our work is based on the lack of support of current feature modeling languages and tools to specify variability in emergent application domains, such as edge computing. We begin the section by contextualizing the language constructs available for feature modeling and use an example in the edge computing domain to illustrate the problematic (Section 2.1). Then, we summarize the current state of the art of feature modeling languages and tools (Section 2.2), focusing on their limitations throughout our example. Finally, we describe the current works trying to cope with these limitations, either by defining a new common variability language or providing interoperability between the existing ones (Section 2.3).

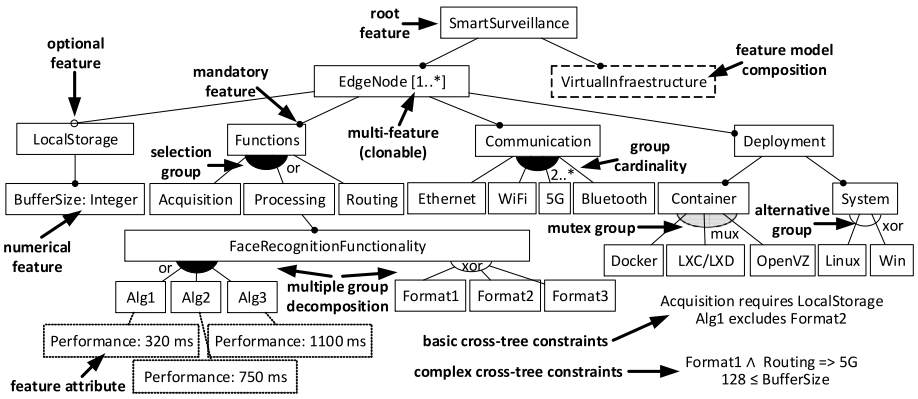


Fig. 1. An excerpt of the feature model of an edge computing system with different types of feature modeling constructs.

2.1. Feature models and language constructs for variability modeling

Features models (FMs) have been widely used to model variability since their introduction in FODA by Kang et al. (1990) and have become the de-facto standard for variability modeling in software product lines (SPLs). FODA (Kang et al., 1990) introduced the basic characteristics for modeling variability in FMs, such as *mandatory* and *optional* features, *alternative* ("xor") and *selection* ("or") groups, and *requires* and *excludes* constraints between features. Due to the success of the FMs for variability modeling, a vast number of modeling languages and extensions have been proposed (ter Beek et al., 2019; Benavides, 2019; Schobbens et al., 2007).

Focusing specifically on the language constructs desirable in nowadays application domains, Fig. 1 illustrates the complexity of modeling the variability of an edge computing application. It does not pretend to be a complete model, but a partial one to illustrate the problems that need to be faced and for which we are proposing a solution in this paper. It models the variability of a smart surveillance edge computing application, where the figure mainly illustrates the variability of the edge nodes that are part of the application. Notice that the different language constructs being used in this feature model are identified with their names in bold and an arrow pointing to the corresponding feature(s).

An edge computing system presents variability at different levels, although Fig. 1 illustrates only the variability of edge nodes (`EdgeNode` feature) and virtual infrastructures (`VirtualInfrastructure` feature). Moreover, for virtual infrastructure we are using the *feature model composition* (Urli et al., 2012) construct that makes reference to another feature model. Focusing then on the edge nodes, an edge computing application will usually have multiple edge nodes, all defined in terms of the same set of features, but configured differently. For instance, with or without a local storage with different buffer size, providing different services that use different algorithms and data formats, or using different communication mechanisms or deployment strategies. In this scenario, *multi-features (clonables)* (Czarnecki et al., 2005) are needed to express this variability in a simple and elegant way, which is to avoid repeating the same feature model subtree for each edge node. Moreover, the number of instances for each clonable needs also to be specified, and this cannot be done without using the *cardinality* associated to multi-features. In our model, the `EdgeNode` feature is a multi-feature with a [1..*] cardinality. Other examples of language constructs that are commonly needed in these systems are *group cardinalities* (Czarnecki et al., 2005), *numerical features* (Munoz et al., 2019), *multiple groups decomposition* (Czarnecki and Eisenecker, 2000) or *feature attributes* (Benavides et al., 2005). For example, an edge node may support several `Communication` mechanisms, but with a minimum of two (*i.e.,* group cardinality is `2..*`). The edge node can be deployed using or not a specific container modeled as a *mutex group* (Berger et al., 2014) (*i.e.,* it allows selecting 0 or 1 of its alternatives). Also, there are many features whose value is not discrete and thus numerical features are needed (*e.g.,* the type of the `BufferSize` feature is `Integer`). And these numerical features may be involved in *complex cross-tree constraints* as for example that the `BufferSize` must be at least of 128 MB (`128 ≤ BufferSize`).

The complexity of modeling and analyzing this variability relies on the lack of existing languages and tools that support all the presented languages constructs as exposed in the following section.

2.2. Languages and tool support for feature modeling and analysis

There are many languages and tools for feature modeling (Bashroush et al., 2017; ter Beek et al., 2019; Horcas et al., 2022; Galster et al., 2014; Horcas et al., 2020). In this section, we focus on those selected in the practical study published in earlier work (Horcas et al., 2022), which corresponds to the feature modeling languages supported in well-known and widely used tools in the SPL community for feature modeling (see Table 1). For these tools, we summarize their language constructs (Table 1), automatic analysis support (Table 2) and interoperability (Table 3).

By observing Tables 1 and 2 the conclusion is that none of these languages provide all the constructs and the analysis support needed in nowadays application domains, such as in the edge computing domain. Moreover, although there is interoperability between some tools (Table 3), this is not enough to easily combine the use of several of these languages in complex scenarios (Berger and Collet, 2019; Horcas et al., 2022; Galster et al., 2014). Such interoperability is provided by means of import/export operation of the concrete syntax between languages of the same expressiveness. Similar results are obtained by Sepúlveda et al. (2016), where authors identify a low level of maturity of existing languages due to the lack of an abstract syntax and low level of expressiveness.

Observing our case study, a practitioner may easily decide to use Clafer (Juodisius et al., 2019) to model the variability of the edge computing systems, since Clafer is the most expressive language currently available (see Table 1). Indeed, Clafer supports the constructs required by our case study like numerical features and multi-features. Then, the user needs to analyze the feature model by checking for inconsistencies, constraint validation, and calculating the degree of variability by determining the number of valid products. But, according to the information presented in Table 2, the automated analysis of a large model, typical of an

Table 1
Language constructs supported by feature modeling tools (ter Beek et al., 2019; Horcas et al., 2022, 2020).

Tools	Optional feat.	Xor group	Or group	Abstract feat.	Mutex-Group	Group Cardinality	Multi decomp.	Multi-feature	Typed feature	Numerical feat.	Feat. attribute	Binding time	Default value	Delta value	Range	Simple const.	Prop. log. const.	First-order const.	Relational expr.	Arithmetic expr.	Type const.	Default const.	Compositions	Conf. reference	Containers	Model version	Multi-views	Configuration	Partial conf.
Glencoe	●	●	●	○	○	●	○	○	○	○	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	◐	○	●	●
SPLOT	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	●	◐	○	○	○	○	○	○	○	○	○	○	●	●
FaMa	●	●	●	○	○	●	○	○	○	○	●	◐	○	○	○	●	◐	○	○	○	○	○	○	○	○	○	○	●	●
Clafer	●	●	●	●	●	●	○	●	●	●	◐	◐	◐	○	◐	●	●	●	●	●	◐	●	◐	◐	◐	○	○	●	●
FeatureIDE	●	●	●	●	○	○	○	○	○	○	◐	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	●	●
pure::variants	●	●	●	●	○	○	○	◐	○	◐	◐	○	○	○	○	●	●	○	◐	◐	○	○	●	◐	◐	○	○	●	●

●: support. ◐: partially support. ○: not support.

Table 2
Current tool support for automated analysis of feature models (Benavides et al., 2010; Horcas et al., 2022).

Tools	Void feat. model	#Products	Dead features	Valid product	All products	Explanations	Refactoring	Optimization	Commonality	Filter	Valid conf.	Atomic sets	False optional	Corrective explan.	Depend. analysis	ECR	Generalization	Core features	Variability factor	Arbitrary edit	Cond. dead feat.	Homogeneity	LCA	Multi-step conf.	Roots features	Specialization	Orthogonality	Redundancies	Variant features	Wrong cardinal.
Glencoe	●	●	●	●	○	●	○	○	●	○	●	●	●	●	●	●	○	●	●	○	●	●	○	●	○	○	●	●	●	●
SPLOT	●	●	●	●	○	○	○	○	●	○	●	○	○	○	○	●	○	●	●	○	○	○	○	●	○	○	●	○	○	●
FaMa	●	◐	●	●	◐	●	○	○	◐	○	◐	○	○	○	●	○	○	◐	◐	○	○	○	○	●	○	○	○	○	●	○
Clafer	●	◐	○	●	◐	○	○	◐	○	○	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
FeatureIDE	●	◐	●	●	◐	●	○	○	○	○	●	●	●	●	●	●	○	●	◐	○	●	○	○	●	○	○	●	●	●	●
pure::variants	●	◐	○	●	○	●	○	○	○	○	●	○	○	●	●	○	○	○	○	○	○	○	○	●	○	○	●	○	○	●

●: support. ○: not support. ◐: support with limitations or not scale for large models.

Table 3
Current state of the tools interoperability for feature models.

Tools \ Formats	Glencoe	SPLOT	FaMa	Clafer	FeatureIDE	pure::variants	DIMACS	SPASS	Guidsl	v.control	SPL Conquerer	CNF txt	CSV txt
Glencoe	●	○	○	○	○	◑	◑	◑	○	◑	○	○	○
SPLOT	○	◑	○	◑⁻	●	○	○	○	○	○	○	○	○
FaMa	○	○	●	○	○	○	○	○	○	○	○	○	○
Clafer	○	◑⁻	○	●	○	○	○	○	○	○	○	○	○
FeatureIDE	○	●	◑	○	●	○	●	○	●	○	◑	◑	○
Pure::variants	○	○	◐⁻	○	○	●	○	○	○	○	○	○	●

●: import and export. ◐: import. ◑: export. ○: not support.
⁻Supported with external plug-ins (Horcas et al., 2022).

edge-based system, cannot be achieved efficiently with Clafer. For instance, Clafer uses an SMT solver that requires to enumerate all products in order to count all valid configurations (Juodisius et al., 2019). Thus, the feature model should be translated to the concrete syntax of another tool or solver, such as Glencoe (Schmitt et al., 2018) which already provides a complete set of solvers, to be able to make the automated analysis of its variability. But, no direct translation (import/export) exists between these two tools (see Table 3). Here the problem is that Glencoe is not as expressive as Clafer since Glencoe does not support advanced variability characteristics like numerical features or multi-features. At this point, a translation in terms of *edits* (Thüm et al., 2009) between language constructs of both languages is needed. Concretely, a *specialization* (Thüm et al., 2009) (*i.e.,* a transformation in which there is some loss of information) is required in order to represent the numerical features in Glencoe, for example, by discretizing the values of the numerical features into Boolean features. Being aware of these limitations, the user decides to limit the analysis of the edge system to a subset of products including only basic Boolean features. After modeling the sub-version of the case study in Glencoe and analyzing its variability, the user needs to connect the variability with the code implementing the features. However, neither Glencoe nor Clafer provides support for automatically generating the code of products from feature configurations. Therefore, the user needs to use an alternative tool such as FeatureIDE (Meinicke et al., 2017). In this case, a translation from Glencoe to FeatureIDE is needed, but none of the tools provides a direct translation to each other (see Table 3). While Glencoe supports the modeling of group cardinalities (Czarnecki et al., 2005), FeatureIDE supports only xor-groups and or-groups (see Table 1). Thus, a similar problem appears again, but in this case, a transformation in terms of *refactorings* (Thüm et al., 2009) (*i.e.,* a transformation without loss of information) exists because the feature modeling language of both tools (Glencoe and FeatureIDE) is equally expressive (Schobbens et al., 2007).

2.3. Related work about interoperability of feature models

The results discussed in Section 2.2 justifies the large number of approaches that are arising toward either the definition of a common variability modeling approach or the improvement of the interoperability between existing languages. Although the definition of a new common language is not the objective of this paper, there are some design principles that are shared in both approaches (Horcas et al., 2020). For this reason, in this section we first present the state-of-the-art on defining a new common variability language, and then we discuss the refactoring and specialization rule-based approaches that exist in the literature.

Definition of a common variability language. Several works, such as Berger and Collet (2019) and Thüm et al. (2019), focus on identifying the language constructs and language levels that a common variability language should include, considering the scenarios that the resulting language should support. Their main purpose is the organization of existing language constructs at different language levels, with different expressiveness, to handle

the complexity of defining a common language. Other works, such as ter Beek et al. (2019), review textual variability modeling languages and identify five dimensions to classify their elements: configurable elements, constraints, configuration support, scalability support, and language characteristics. Sepúlveda et al. (2016), Sepúlveda et al. (2012) propose a metamodel that includes a set of core language constructs that are identified after a systematic review of the maturity and expressiveness level of existing variability modeling languages. *These goals of classifying the variability modeling elements and of modeling the language constructs in different levels are shared with the modular definition of our common abstract syntax, in which our interoperability approach is based. In fact, the levels of expressiveness and language constructs identified in these works were the starting point for the definition of the common ground in our approach.*

The definition of the new common variability language in a modular way is shared by many existing works (Benavides et al., 2019). Thanks to the *MODEVAR* initiative (Benavides et al., 2019), there is currently an attempt of a *universal variability language* (UVL) (Sundermann et al., 2021) proposed by the SPL community. The goal of UVL is to be as simple as possible and to cover the needs of current demanding requirements, but, actually, UVL is in its infancy and only supports basic variability modeling. Seidl et al. (2016) define an SPL of feature modeling notations and constraints to generate different variants of them. Villota et al. (2019) introduce the *high-level variability language* (HLVL), a unified language that follows an orthogonal approach and serves as an intermediate language for variability. Butting et al. (2018a,b) propose the definition of a family of domain-specific modeling languages that focus on reusing abstract syntax in the form of metamodel parts. Their approach composes the syntax and semantics of independently developed modeling languages through the use of a composition mechanism based on well-defined language extension points. Zhiyi and Xiao (2014) build a family of software modeling tools based on metamodeling and SPL technologies. A feature model specifies the commonality and variability of modeling tools and provides a general tool framework for reusing components and generating code for components, specifying the mapping between the feature model and the components for modeling tools. *All these works share with our proposal the necessity of having a modular approach. More concretely, we share with them Butting et al. (2018a,b), Zhiyi and Xiao (2014) the use of abstract syntax and metamodeling. The difference is that they focus on the definition of new languages, and our proposal focuses mainly on achieving interoperability between existing languages. In any case, our modular and abstract syntax could also be used for the definition of a new language.*

Feature modeling languages interoperability. A more feasible alternative to the definition of new variability modeling languages is to improve the interoperability among existing ones. Interoperability and translations between feature modeling languages are achieved by the definition of edits (Thüm et al., 2009) and model transformations (Feichtinger, 2021; Feichtinger and Rabiser, 2020). In Thüm et al. (2009), four possible edits are identified: (1) a *feature-model refactoring* preserves exactly the set of valid feature selections; (2) a *specialization* removes feature selections without adding new ones; (3) a *generalization* adds valid feature selections without removing any; and (4) an *arbitrary edit* both adds and removes valid feature selections. Also, Alves et al. (2006) propose a set of unidirectional refactorings, but only between basic constructs (*e.g.,* from an "or" feature to an optional feature, from optional to mandatory). Tanhaei et al. (2016) automate those refactorings through a model transformation approach, while Gheyi et al. (2011) automate the process of checking the refactorings validation. However, these works (Alves et al., 2006; Gheyi et al., 2011; Tanhaei et al., 2016) do not apply the transformations among different expressiveness or language levels, dealing only with basic variability modeling. Recently, Feichtinger (2021), Feichtinger and Rabiser (2021) study how flexible a transformation approach for variability models must be, and propose TRAVART (Feichtinger and Rabiser, 2020; Feichtinger et al., 2021), an approach transforming artifacts describing variability. TRAVART focuses on transformations between different variability modeling approaches such as feature modeling, decision modeling, and orthogonal variability modeling (OVM); in contrast to our approach where we focus on a specific variability modeling approach: feature models, and thus, we deal with transformations between constructs of feature modeling concepts.

Horcas et al. (2022) define roadmaps for tool interoperability based on the activity to be performed (*e.g.,* modeling, analysis, derivation), but these roadmaps are limited to the current import/export support of the tools (Table 3). Concerning language analysis, Schobbens et al. (2007) provide formal semantics for feature diagrams that result in a language: *varied feature diagrams* (VFD) that is expressively complete in terms of Boolean features, making it able to express several diverse constructs. Knüppel et al. (2017) analyze whether less expressive languages are sufficient for industrial SPLs. They focus on feature models with complex constraints and provide an algorithm to express these constraints using only basic feature models. Romero et al. (2021) propose a new repository for feature model exchange. They list information that would be useful to store in the repository, as well as 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 enables the interoperability of large-scale case studies, by defining a common ground where reusable functionality (*e.g., *model transformations) can be defined based on notation-independent language constructs and shared across existing languages.*

3. Our approach to achieve SPL interoperability

Fig. 2 shows our approach to provide interoperability between existing feature modeling languages. An essential characteristic of our work is that it relies on the idea of having a common, general, and modular abstract syntax for feature modeling (called $\mathbb{CAF}$), which incrementally includes all the important language constructs to model variability in complex application domains such as edge computing, video processing, or cyber–physical systems. Concretely, we propose an approach based on model-driven engineering (MDE) and metamodeling (Asikainen and Männistö, 2009; Atkinson and Kuhne, 2003) that applies the concepts of extensible language (ter Beek et al., 2019) and modular language design (Thüm et al., 2019) to specify the abstract syntax of different language constructs. The core element is a set of extensible and modular metamodels which specify the abstract syntax of all language constructs from existing languages (middle-top of Fig. 2). Details of these metamodels are presented in Section 4.

The other key component of our approach is that, using this common abstract syntax, it is possible to define reusable mappings among language constructs with similar expressiveness. This mapping is independent of the concrete syntax or internal representation used in each language. Moreover, the definition of the mappings at the abstract syntax level makes them reusable in the mapping between any pair of concrete languages. These mappings are represented as model-to-model (M2M) transformations defined at the language construct level. Concretely, our approach supports the four kinds of model transformations identified by Thüm et al. (2009) as *edits* for feature models: *refactorings*, *specializations*, *generalizations*, and *arbitrary edits*. In this paper, we are interested in refactorings, which are those that do

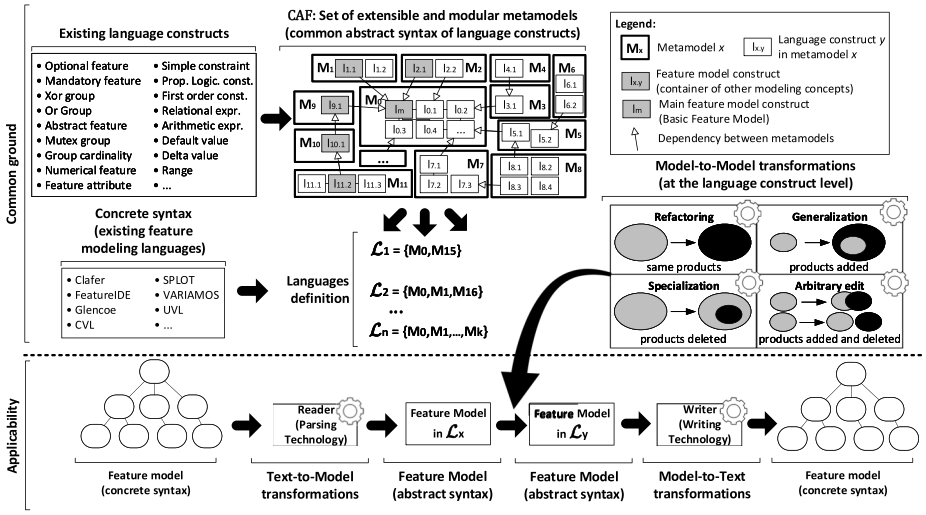


Fig. 2. An extensible MDE approach to provide interoperability between feature modeling languages guided by the abstract syntax of the language constructs.

not lose information between transformations. Details of these model transformations are given in Section 5.

The common abstract syntax (CAF) and the M2M transformations are the core of our approach. However, for this approach to be adopted, it is important that domain engineers can continue using the feature modeling languages they are used to. This means that, as illustrated in Fig. 2, to represent a feature model with our abstract syntax, we first need to deal with the concrete syntax of its language (*e.g.*, Clafer, FeatureIDE). This is achieved by using text-to-model (T2M) transformations from the concrete syntax to our abstract syntax. Such T2M transformations are usually implemented using parsing technologies such as *ANTLR*.[1] As a result, we obtain a feature model represented using the language constructs of a specific subset $\mathcal{L}_x$ of the metamodels in CAF. Once the languages are represented in our abstract syntax, the generic mapping between them can be done in terms of the mapping between the language constructs used in each language to represent similar variability modeling concepts. As a result, we obtain a feature model represented using the language constructs of the language $\mathcal{L}_y$. Finally, to translate from the abstract syntax to the concrete syntax of each language, a set of model-to-text (M2T) transformations can be defined using a writing or generator technology.

In the following sections we detail the two main parts of our approach: the common abstract syntax (Section 4) and the M2M transformations (Section 5) between language constructs.

4. CAF: Common abstract syntax for feature modeling

In previous work (Horcas et al., 2020), the abstract syntax of the language constructs for feature modeling was formalized as a set of extensible and modular metamodels (called FM). For self-containment, we summarize and illustrate this formalization, renaming it as CAF (Section 4.1). Then, we focus on showing a specific realization of CAF with a subset of existing language constructs from the literature (Section 4.2), and on formally demonstrating that by combining several of the existing language constructs defined in CAF, we are able to represent the semantics of existing feature modeling languages (Section 4.3).

[1] https://www.antlr.org/

4.1. Formalization of CAF

We use the following definitions for feature modeling (Harel and Rumpe, 2004; Batory, 2005; Schobbens et al., 2007):

Definition 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 subset 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.*, $spl \in \mathcal{P}(\mathcal{P}(P))$.

Any modeling language must consist of three elements (Harel and Rumpe, 2004): a *syntactic domain* ($\mathcal{L}$), a *semantic domain* ($\mathcal{S}$), and a *semantic function* ($\mathcal{M}$). Fig. 3 illustrates these concepts. 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. The *concrete syntax* is the physical representation of the feature model in terms of textual or graphical notations (*e.g.*, the feature diagram). For convenience, we use interchangeably $\mathcal{L}$ as the abstract syntax and the syntactic domain in the following. 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 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.

The semantic function $\mathcal{M}$ is *total* and is defined for all elements of $\mathcal{L}$. This 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 (*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 $E(\mathcal{L}_2) \subset E(\mathcal{L}_1)$.

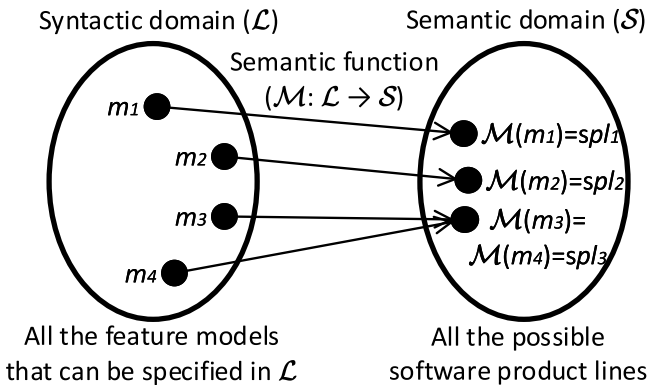


Fig. 3. Syntactic domain, semantic domain, and semantic function for feature modeling.
Source: Adapted from Heymans et al. (2007).

One of the mechanisms for expressing the constructs of a modeling language, as well as their relationships and constraints, is a *metamodel* (Atkinson and Kuhne, 2003; Asikainen and Männistö, 2009). A *metamodel* (Atkinson and Kuhne, 2003; Asikainen and Männistö, 2009) 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 2. The metamodel is specified using a metalanguage, *e.g.,* Meta-Object Facility (MOF) (Object Management Group (OMG), 2016).

Definition 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 $\mathcal{M}[\![m]\!] \neq \emptyset$.

Given a well-formed 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 $m \in \mathcal{L}_M$.[2] 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 feature modeling concept (*e.g.,* optional feature, group feature, requires constraint, multi-feature, attributed feature).

Definition 5 (*Language Construct*)**.** A language construct $l \in M$ is the abstract syntax of a specific feature 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 (Czarnecki et al., 2005), `NumericalFeature` for non-Boolean numerical features (Munoz et al., 2019), `FeatureAttribute` for attributed features (Benavides et al., 2005), `BindingTime` to model the points in time when the variability may happen (ter Beek et al., 2019), and so on. Here, we also define a special language construct for feature models:

Definition 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 feature modeling concepts such as features and constraints. That is, l_m is the main containment element in the metamodel (*aka*, the root element).[3]

Examples of feature model constructs are `Feature Model` to represent the most generic feature model, `Cardinality-Based FM` to represent feature models with cardinalities (Czarnecki et al., 2005), `Attributed FM` for models that support features with attributes (Benavides et al., 2005), `Numerical FM` for models with numerical features (Munoz et al., 2019), etc.

With these definitions, we define $\mathbb{CAF}$ as a set of modular metamodels as follows:

Definition 7 ($\mathbb{CAF}$)**.** $\mathbb{CAF}$ is a set of inter-related metamodels, *i.e.,* $\mathbb{CAF} = \{M_0, M_1, M_2, \ldots, M_n\}$, where each metamodel $M_i \in \mathbb{CAF}$ is a different non-empty set of language constructs, *i.e.,* $M_i = \{l_1, l_2, \ldots, l_t\}$ and $M_i \cap M_j = \emptyset, \forall M_i, M_j \in \mathbb{CAF}, i \neq j$.

The criteria to decide whether a language construct is defined in one metamodel or another is a design decision and is part of the realization of $\mathbb{CAF}$ (Section 4.2).

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

Definition 8 (*Extension, Composition, Dependency*)**.** 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 $M_i \in \mathbb{CAF}$ *depends* on another metamodel $M_j \in \mathbb{CAF}$, 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). Note that $<:$ is transitive, *i.e.,* $\forall l_i, l_j, l_k : l_i <:_j \wedge l_j <: l_k \Rightarrow l_i <: l_c$. 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. We do not impose any restriction in $\mathbb{CAF}$ regarding the arity of those relationships, and thus, $\mathbb{CAF}$ allows a language construct to extend more than one language construct (*i.e.,* multiple inheritance), and to be composed by more than one language construct (*i.e.,* multiple composition) being defined in the same or distinct metamodels.

To complete the definition of $\mathbb{CAF}$ let us define an initial metamodel $M_0 \in \mathbb{CAF}$ with, at least, a feature model construct $l_m \in M_0$ which describes the generic concept of feature model (Definition 6). 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 2, that is, the set of products that can be specified with those language constructs. The boxes in the middle-top of Fig. 2 show a generic schema of $\mathbb{CAF}$ that illustrates all the concepts formalized. In the following, we present a concrete realization of $\mathbb{CAF}$ with existing language constructs for feature modeling.

4.2. Realization of $\mathbb{CAF}$ with existing feature modeling constructs

$\mathbb{CAF}$ can be realized with any number of metamodels and language constructs as long as the structure of $\mathbb{CAF}$ proposed in the previous section is respected. Ideally, $\mathbb{CAF}$ should expose

[2] We overload $\in$ for set membership and feature models specified in a language $\mathcal{L}$.

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

Table 4
Realization of $\mathbb{CAF}$ with existing language constructs for feature modeling extracted from the literature (part I: configurable elements). We show its metamodels (M), the set of language constructs, their extension relationships of each language construct (Ext.), the metamodel dependencies (Dep.), and the semantics and main references.

M	Language constructs	Ext.	Dep.	Semantic	Refs.
	CONFIGURABLE ELEMENTS				
M_0	l_{m}: *Feature Model*	–	–	The top element of the metamodel.	Kang et al. (1990)
	$l_{0,1}$: *Feature*	–	–	The unit of variability. A feature can be *optional* or *mandatory*.	
	$l_{0,2}$: *Root*	$l_{0,1}$	–	The root feature $r \in F$ of the feature model. r is always mandatory.	
	$l_{0,3}$: *Optional Feature*	$l_{0,1}$	–	It represents an *optional* feature.	
	$l_{0,4}$: *Mandatory Feature*	$l_{0,1}$	–	It represents a *mandatory* feature.	
	$l_{0,5}$: *Parent-Child Rel.*	–	–	Features decomposition. A child can be selected only when its parent is selected.	
	$l_{0,6}$: *Feature Group*	$l_{0,1}$	–	Children of a feature $f \in F$ can be grouped.	
	$l_{0,7}$: *Alternative Group*	$l_{0,6}$	–	It defines a one-out-of-many choice, *i.e.,* an *xor group* <1..1>.	
	$l_{0,8}$: *Or Group*	$l_{0,6}$	–	It defines a some-out-of-many choice, *i.e.,* an *or group* <1..*>.	
	$l_{0,9}$: *Cross-Tree Constr.*	–	–	The generic concept of a constraint.	
M_1	$l_{1,1}$: *Abstract Feature*	$l_{0,1}$	M_0	Distinction between concrete and abstract features in leaf features.	Knüppel et al. (2017)
M_2	$l_{2,1}$: *Mutex-Group*	$l_{0,6}$	M_0	Feature groups where at most one feature can be selected.	Berger et al. (2013)
M_3	$l_{3,1}$: *Cardinality-Based FM*	l_{m}	M_0	It allows defining cardinalities for features and groups.	Czarnecki et al. (2005)
	$l_{3,2}$: *Multiplicity*	–	–	It defines lower and upper bounds.	
M_4	$l_{4,1}$: *Group Cardinality*	$l_{0,6}$	M_0, M_3	Arbitrary multiplicities <n..m> for group features, bounded and unbounded (*).	Czarnecki et al. (2005).
M_5	$l_{5,1}$: *Multiple Decomp.*	$l_{0,5}$	M_0	Different group features (*e.g.,* or and xor), below the same feature.	Czarnecki and Eisenecker (2000)
M_6	$l_{6,1}$: *Dir. Acyclic Graph*	$l_{0,5}$	M_0	Features with multiple parents.	Kang et al. (1998)
M_7	$l_{7,1}$: *Multi-Feature*	$l_{0,1}$	M_0, M_3	Features with cardinalities (*aka*, clonable features).	Czarnecki et al. (2005)
M_8	$l_{8,1}$: *Non-Boolean FM*	l_{m}	M_0	Definition of arbitrary data types for features and/or attributes.	Juodisius et al. (2019)
	$l_{8,2}$: *Data Type*	–	–	Primitive (*e.g.,* Boolean, Integer, Float, String,...), and user-defined types.	
	$l_{8,3}$: *Value Assignment*	–	–	It allows providing a value to a specific data type.	
	$l_{8,4}$: *Typed Feature*	$l_{0,1}$	–	Arbitrary data types for features.	
	$l_{8,5}$: *Attached Inf.*	–	–	Additional information (e.g., attributes, meta-attributes) for configurable elements.	
M_9	$l_{9,1}$: *Numerical FM*	$l_{8,1}$	M_0, M_8	Feature model with numerical features.	Munoz et al. (2019)
	$l_{9,2}$: *Numerical Feature*	$l_{8,4}$	–	Non-Boolean numerical features (*e.g.,* Natural, Integer, Real,...).	
M_{10}	$l_{10,1}$: *Attributed FM*	$l_{8,1}$	M_0, M_8	Feature models with attributes.	Benavides et al. (2007)
	$l_{10,2}$: *Feature Attribute*	$l_{8,5}$		Features with attributes (*e.g.,* cost, performance).	
M_{11}	$l_{11,1}$: *Binding time*	$l_{8,5}$	M_0, M_8	Point time when the variability decision must be made.	Schmid et al. (2018)
M_{12}	$l_{12,1}$: *Default Value*	$l_{8,3}$	M_0, M_8	It allows establishing a default value to a typed feature or attribute.	Al-Azzawi (2018)
M_{13}	$l_{13,1}$: *Delta Value*	$l_{8,3}$	M_0, M_8, M_9, M_{18}	It reduces the number of acceptable numeric values.	Alférez et al. (2019)
M_{14}	$l_{14,1}$: *Range*	$l_{8,3}$	M_0, M_8, M_9, M_{18}	It allows defining ranges of values for numerical features or attributes.	Alférez et al. (2019)

the language constructs of existing feature modeling languages, the semantics of most of them have been already formalized in Schobbens et al. (2007), Eichelberger et al. (2013). Tables 4 to 7 present the complete realization of $\mathbb{CAF}$ (adapted from Horcas et al. (2020)) with existing language constructs from the literature. For each metamodel, we present its set of language constructs, the main relationship between the language constructs (the *extends* relation, *aka*, the supertype), and the metamodel dependencies. We also show the semantics and the main references for the language constructs introduced in each metamodel. The modularization of the language constructs is based on different criteria. First, metamodels and language constructs have been classified following four of the dimensions proposed in ter Beek et al. (2019): configurable elements (Table 4), constraints support (Table 5), scalability support (Table 6), and configurations (Table 7). Second, we classify the language constructs based on the type of variability they model. For instance, metamodel M_0 contains the FODA (Kang et al., 1990) 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 (Cordy et al., 2013) such as numerical features (Munoz et al., 2019), attributes (Benavides et al., 2005), or additional information (Alférez et al., 2019). Third, we put together in the same metamodel those language constructs whose definitions directly depend on other constructs (Definition 8). For example, the `Typed Feature` construct, which represents arbitrary data types for features, requires to define a data type (`Data Type` construct) and providing a value (`Value Assignment` construct). Thus, we define all these constructs together in metamodel M_8. However, this is not a strict rule as explained below for the `Group Cardinality` and `Multi-Feature` constructs.

The realization exposes more than 50 language constructs over more than 30 metamodels. Despite the high number of

Table 5
Realization of $\mathbb{CAF}$ (part II: constraints support).

M	Language constructs	Ext.	Dep.	Semantic	Refs.
			CONSTRAINTS SUPPORT		
M_{15}	$l_{15,1}$: *Simple dependency*	$l_{0,9}$	M_0	*Requires* and *excludes* constraints.	Kang et al. (1990)
M_{16}	$l_{16,1}$: *Propositional logic*	$l_{0,9}$	M_0	Arbitrary propositional formulas over the features ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$).	Batory (2005)
M_{17}	$l_{17,1}$: *First-order logic*	$l_{0,9}$	M_0, M_3, M_7	Quantifiers ($\forall$, $\exists$), predicates, functions, and constants.	Asikainen et al. (2006)
M_{18}	$l_{18,1}$: *Relational expr.*	$l_{0,9}$	M_0, M_8	Operators for comparing features or attributes ($==$, $<$, $<=$, $>$, $>=$, $\neq$, !).	Al-Azzawi (2018)
M_{19}	$l_{19,1}$: *Arithmetic expr.*	$l_{0,9}$	M_0, M_8	Arithmetic formulas, functions, and operators ($+$, $-$, $\times$, /, %,...).	Al-Azzawi (2018)
M_{20}	$l_{20,1}$: *Cardinality expr.*	$l_{0,9}$	M_0, M_3, M_4	Cardinalities expressed in terms of constraints.	Czarnecki et al. (2005)
M_{21}	$l_{21,1}$: *Type restrictions*	$l_{0,9}$	M_0, M_8	Type-specific operators for constraints (*e.g.,* String operators, regular expressions).	Haugen et al. (2008)
M_{22}	$l_{22,1}$: *Default constraints*	$l_{0,9}$	M_0	Constraints that can be altered as part of the constraint-resolution process.	Schmid et al. (2018)

Table 6
Realization of $\mathbb{CAF}$ (part III: scalability support).

M	Language constructs	Ext.	Dep.	Semantic	Refs.
			SCALABILITY SUPPORT		
M_{23}	$l_{23,1}$: *Compositional FM*	l_m	M_0, M_8	Mechanisms for composition and inheritance for large feature models.	Acher et al. (2013)
	$l_{23,2}$: *Interface*	–	–	The concept of interface of feature model for modularization.	
	$l_{23,3}$: *Scope*	–	–	Support for declaring scopes (*e.g.,* scoped import of models).	
M_{24}	$l_{24,1}$: *Configuration Refer.*	$l_{23,2}$	M_0, M_{23}	It defines links between models and configurable elements.	Haugen et al. (2008)
M_{25}	$l_{25,1}$: *Containment Feature*	$l_{23,2}$	M_0, M_{23}	Composition on type level (*aka,* composite units).	Haugen et al. (2008)
M_{26}	$l_{26,1}$: *Imports*	–	M_0, M_{23}	Import of models.	Schmid et al. (2018)
M_{27}	$l_{27,1}$: *Merge*	–	M_0, M_{23}	Operator for overlapping.	Acher et al. (2013)
M_{28}	$l_{28,1}$: *Aggregate*	–	M_0, M_{23}	Operator for disjoint models.	Acher et al. (2013)
M_{29}	$l_{29,1}$: *Include*	–	M_0, M_{23}	Models can be composed of a model of a larger scale.	Classen et al. (2011)
M_{30}	$l_{30,1}$: *Model Version*	$l_{8,1}$	M_0, M_8	Support for managing versions of the models.	Schmid et al. (2018)
M_{31}	$l_{31,1}$: *Visibility*	$l_{8,1}$	M_0, M_8	Visibility (*e.g.,* public, private) for configurable elements.	Abele et al. (2010)
M_{32}	$l_{32,1}$: *View-Points*	$l_{8,1}$	M_0, M_8	Multiple view-points for feature models.	Rosenmüller et al. (2011)

Table 7
Realization of $\mathbb{CAF}$ (part IV: configurations).

M	Language constructs	Ext.	Dep.	Semantic	Refs.
			CONFIGURATIONS		
M_{33}	$l_{33,1}$: *FM Configuration*	–	M_0	A full configuration of a feature model as a selection (and assignment) of features.	Kang et al. (1990)
M_{34}	$l_{34,1}$: *FM Partial Config.*	$l_{33,1}$	M_0, M_{33}	A partial configuration of a feature model.	Alférez et al. (2019)

metamodels and language constructs, the modular design of $\mathbb{CAF}$ allows practitioners to use just the parts they are interested in, together with those parts that are needed to generate a valid language, as explained in the following section. 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 including 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, allows the automatic selection of the appropriate language constructs to define a specific language with the desired modeling concepts. Fig. 4 illustrates graphically the realization of $\mathbb{CAF}$ with a subset of the metamodels defined in Tables 4 to 7.

4.3. Definition of existing feature modeling languages using $\mathbb{CAF}$

We can combine the language constructs of different metamodels to represent existing languages with different level of expressiveness. The selection of the language constructs cannot be done arbitrary and needs to be coherent in order to represent a language where all the constructs make sense. That is, we need to respect the metamodels' relationships, as exposed in Tables 4 to 7. Otherwise, the language represented by the selected metamodels may be unexpressive or senseless. For instance, the relational and arithmetic constructs to specify constraints expressions over numerical values (metamodels M_{18} and M_{19}) require to select non-Boolean feature models (M_8), since M_8 defines the modeling concepts to be used within those types of constraints (*e.g.,* data types and value assignment). However, neither numerical (M_9) nor attributed feature models (M_{10}) are explicitly

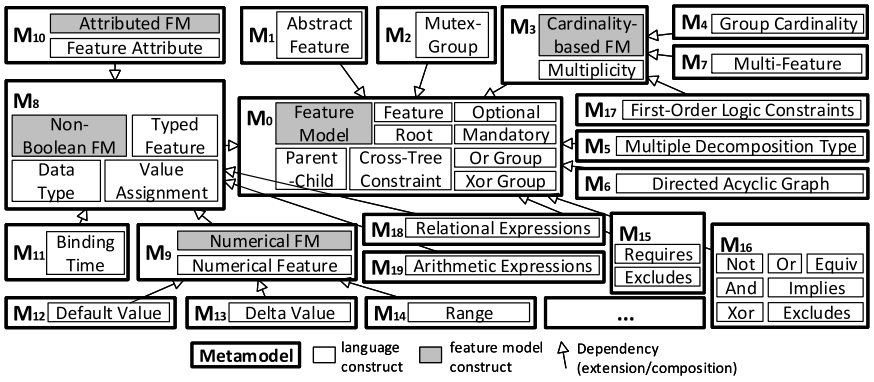


Fig. 4. Realization of $\mathbb{CAF}$ (excerpt from metamodels in Tables 4 to 7).

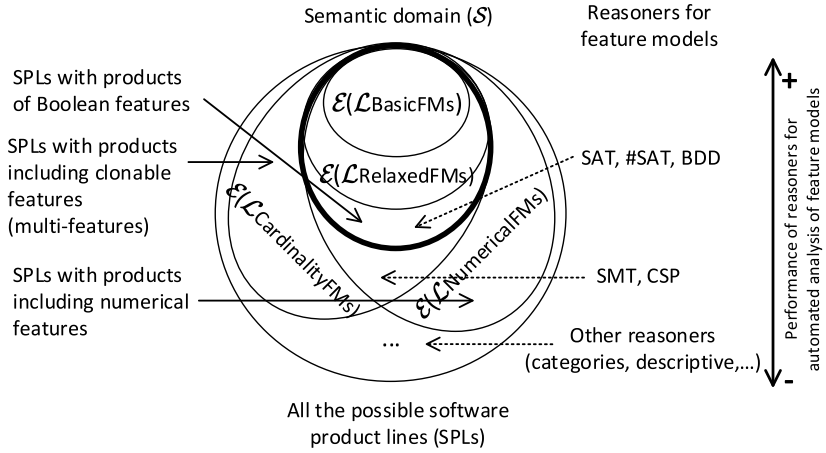


Fig. 5. Trade-off between the expressiveness of existing feature modeling languages and the support for automated reasoning on them.

required, because numerical constraints can be applied over both numerical features and attributes independently, and we let practitioners choose them based on their needs. We illustrate this with four feature modeling languages taken from real-world that are well-known in the community and are representative of the different levels of expressiveness according to the language constructs they offer (Fig. 5). Here we focus on those language constructs that affect the expressiveness of the language (Definition 4), that is, the configurable elements (Table 4) and the constraints support (Table 5).

Basic feature models (the FODA language). Consider the language constructs specified in metamodels M_0 and M_{15} as shown in Fig. 4. The union of those language constructs defines the abstract syntax for basic feature modeling, that is, mandatory and optional features, alternative ("xor") and "or" group features, and *requires* and *excludes* cross-tree constraints. We denote this language $\mathcal{L}_{BasicFMs}$ defined as: $\mathcal{L}_{BasicFMs} = \mathcal{L}_{M_0 \cup M_{15}}$.

This language corresponds with the abstract syntax of the FODA language (Kang et al., 1990) and it is supported by all existing tools for feature modeling (Horcas et al., 2022). However, $\mathcal{L}_{BasicFMs}$ is expressively incomplete since there are products of certain SPL, which cannot be defined with their language constructs (Knüppel et al., 2017) (see Fig. 5), and more complex constraints are needed, as for example those defined in M_{16} to support propositional logic.

Relaxed feature models. Knüppel et al. (2017) demonstrated that complex constraints, such as those defined in M_{16}, can be represented with a simpler set of language constructs, as long as the language (*e.g.*, $\mathcal{L}_{BasicFMs}$) allows for leaf features to be abstract. That is, the language needs to include the abstract feature construct (defined in M_1). Knüppel et al. (2017) call this language *Relaxed Feature Models*, and it is defined in our approach as: $\mathcal{L}_{RelaxedFMs} = \mathcal{L}_{M_0 \cup M_1 \cup M_{15}}$.

The problem with $\mathcal{L}_{RelaxedFMs}$ is that the use of this language may result in degenerated feature models with a loss of structure (*i.e.*, lack of *naturalness* of the feature model (Schobbens et al., 2007)) or an exponential increase in size (*i.e.*, lack of *succinctness* of the model (Schobbens et al., 2007)). This occurs when the language is not *embeddable* (Schobbens et al., 2007), that is, when not all language constructs of the source language can be directly translated to a similar language constructs of the target language, and a combination of constructs is required to express the same modeling concept.

Most of the existing tools (*e.g.*, FeatureIDE, Glencoe) support, at least, the $\mathcal{L}_{BasicFMs}$ or the $\mathcal{L}_{RelaxedFMs}$ language and, also, the propositional logic constraints (M_{16}) that facilitate the modeling of cross-tree constraints and avoid complicated models in terms of design and structure. But, current application domains require modeling more complex variability beyond Boolean features as shown in Section 2.

Numerical feature models. Let us consider an SPL for our edge computing domain that contains configurable parameters that require a numerical value in order to fully configure a product (*e.g.*, the buffer size or the data capturing frequency of the sensor devices). In order to represent a language supporting numerical feature models ($\mathcal{L}_{NumericalFMs}$), we need the language constructs defined in metamodel M_9 (see Fig. 2). However, M_9 by itself is not expressive enough because it only defines the concept of

numerical feature, and it needs support for defining types and assigning values to the numerical features (dependency with M_8), in addition to the concepts for basic feature modeling (dependency with M_0). Thus, $\mathcal{L}_{NumericalFMs}$ will require, at least, the language constructs defined in metamodels M_0, M_8, and M_9. Moreover, if we want to express constraints over the numerical features such as relational or arithmetic expressions we will also need metamodels M_{18} and M_{19}, respectively.

There are few real languages that currently support numerical features. The main reason is that numerical features highly complicate the automating reasoning of the models, requiring an SMT or CSP solver instead of a SAT solver to deal with arithmetic and relational expressions (Galindo et al., 2019). Two examples of languages supporting numerical features are Clafer (Juodisius et al., 2019) and CVL (Haugen et al., 2008).

Cardinality-based feature models. Cardinality-based feature models were introduced by Czarnecki et al. (2005) to mainly support two characteristics: (1) group cardinalities, which allow specifying arbitrary multiplicity in feature groups; and (2) features with cardinalities (*aka*, clonable features) which allow instantiating multiple times the subfeatures of the clonable feature. These characteristics enable the possibility to define advanced constraints such as first-order logic (M_{17}) and cardinality expressions (M_{20}). Cardinality-based feature models can be represented in $\mathbb{CAF}$ as: $\mathcal{L}_{CardinalityFMs} = \bigcup M_i, \forall i \in \{0, 1, 3, 4, 7, 16, 17, 20\}$.

Following this approach, we can represent any language, as long as its constructs are part of the common abstract syntax defined in the metamodels of $\mathbb{CAF}$. Fig. 5 shows the trade-off between the expressiveness of feature modeling languages and the type of reasoners that support each language. While those languages that represent SPLs with products of Boolean features ($\mathcal{L}_{BasicFMs}$ and $\mathcal{L}_{RelaxedFMs}$) can be efficiently analyzed with SAT (Liang et al., 2015), #SAT (Sundermann et al., 2020), and BDD (Heradio et al., 2016) solvers, complex languages representing SPLs with products including clonable features ($\mathcal{L}_{CardinalityFMs}$) or numerical features ($\mathcal{L}_{NumericalFMs}$), require other types of reasoners such as SMT or CSP solvers (Benavides et al., 2010), that are usually less efficient in performing analysis operations on feature models (*e.g.,* counting configurations). To close this gap, our approach enables model-to-model transformations at the language construct level.

5. Model transformations between language constructs

$\mathbb{CAF}$ allows defining model transformations between the abstract syntax of the language constructs in order to represent the same expressiveness of a given feature model with a different set of language constructs. Since transformations in our approach are done at the language construct level, a refactoring in our approach is possible when there exist language constructs in both the target and the source languages that are equally expressive. Otherwise, we can specify a specialization where, despite some loss of information, the resulting feature models representing a subset of the SPL can be efficiently analyzed with existing reasoners (*e.g.,* SAT, #SAT, or BDD solvers).

In the SPL literature, the language constructs for which have been demonstrated to exist a refactoring, belong to transformations between $\mathcal{L}_{CardinalityFMs}$ and $\mathcal{L}_{RelaxedFMs}$. Concretely, those refactorings illustrated in Fig. 6 formalized by Knüppel et al. (2017):

1. **Complex propositional logical cross-tree constraints** ($M_{16} \rightarrow M_0 \bigcup M_1 \bigcup M_{15}$). Cross-tree constraints defined using complex propositional logic formulae can be transformed into basic constraints using requires and excludes constraints (M_{15}), and abstract trees (M_0 and M_1) composed together with the original feature model (see first refactoring in Fig. 6).
2. **Multiple Group Decomposition** ($M_5 \rightarrow M_0 \bigcup M_1$). Multiple group decompositions (*e.g.,* an alternative group and an or-group below the same feature), can be eliminated by substituting each group by a mandatory abstract feature (second refactoring in Fig. 6).
3. **Mutex-group** ($M_2 \rightarrow M_0 \bigcup M_1$). Mutex-group (*i.e.,* groups where at most one feature can be selected) are transformed by adding an optional abstract sub-feature that becomes an alternative group (third refactoring in Fig. 6).
4. **Group cardinality** ($M_4 \rightarrow M_0 \bigcup M_{16}$). Finally, group cardinalities are transformed by changing its sub-features to optional features and adding a complex cross-tree constraint with the allowed combination of features (last refactoring in Fig. 6).

While languages using these language constructs have been demonstrated to be as expressive as $\mathcal{L}_{RelaxedFMs}$, other language constructs such as clonable features (M_7) or numerical features (M_9) have not been proven that can be represented with basic language constructs because of the presence of the advanced constraints such as those introduced in metamodels M_{17-20}. For example, a numerical feature language construct may be transformed to Boolean features by discretizing the possible values that the numerical feature can take (see Fig. 7). However, since for instance the numerical feature `BufferSize` in our example can take any integer value larger than 128, not all values can be represented with Boolean features and there are products that cannot be represented by the new feature model. Moreover, numerical features, like feature attributes language constructs may require to take into account several decisions when formalizing and defining their M2M transformations, such as deciding the range of values to be considered when the cross-tree constraints do not limit the values, deciding the precision to be considered in case of real numbers instead of integers, or considering complex constraints where numerical values are involved together with propositional logic constraints. Defining a refactoring for this case can be complicated and in some case not possible. In such cases, enabling the possibility of defining a specialization is appropriate. Specializations make sense when we are interesting in analyzing and testing specific subsets of products in an SPL.

The other kind of transformations (generalization and arbitrary edits) are also important and part of our approach. In contrast to specializations, generalizations increase the number of products of the resulting feature model when applying the M2M transformation. This may happen when the target language construct is more expressive than the source language construct and the configuration space is not well restricted with the cross-tree constraints. For example, a numerical feature that cannot be limited with cross-tree constraints (*e.g.,* maybe because the target language does not support constraints involving arithmetic expressions). Finally, arbitrary edits modify the set of products of the feature model by adding new products and removing existing ones. For example, changing an alternative group by an or-group or other way round. As exposed by Thüm et al. (2009), designers should avoid arbitrary edits and restructure them in terms of a sequence of specializations, generalizations, and refactorings, to understand the evolution of the feature model.

6. Proof of concepts and validation

We apply our approach to enable interoperability between existing tools and implemented a prototype of $\mathbb{CAF}$ and the refactorings to answer the following research questions (RQs):

- **RQ1:** *How expressive are the feature modeling languages of the existing tools?* **Rationale:** In order to provide interoperability we need to know the level of expressiveness of

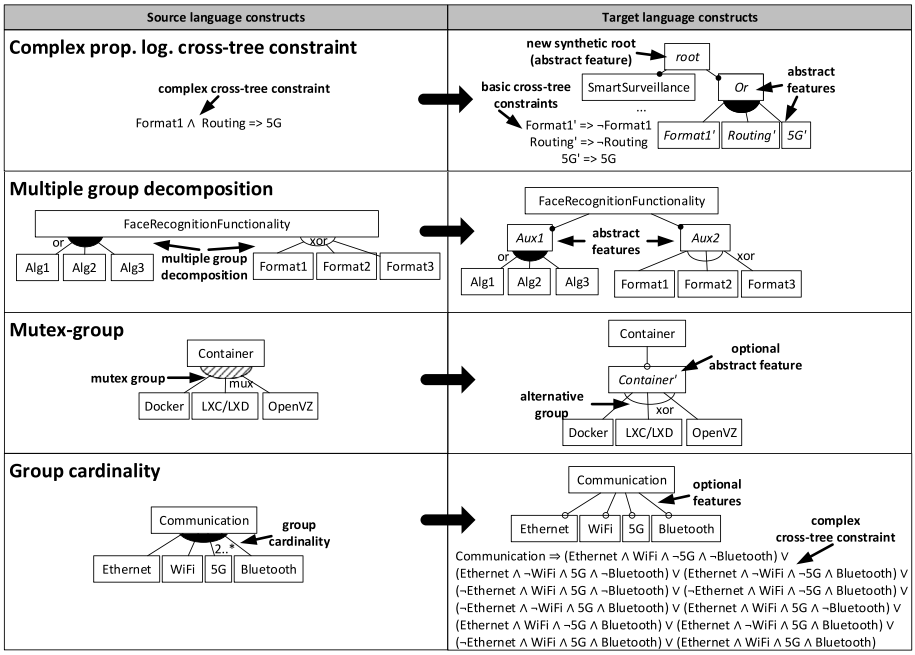


Fig. 6. Refactorings between language constructs formalized in Knüppel et al. (2017).

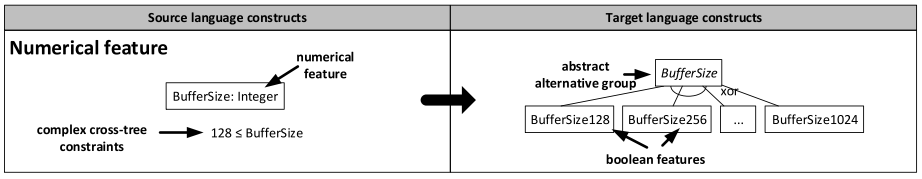


Fig. 7. Example of specialization with loss of information (*i.e.*, existing products are removed within the transformation).

the languages involved. We map the languages of existing tools to the abstract syntax of $\mathbb{CAF}$ and compare their expressiveness with the dialects modeled in Section 4.3.

- **RQ2:** *What is the complexity of providing interoperability between existing feature modeling languages?* **Rationale:** The complexity can be measured by the number of transformations required to provide such interoperability independently of the approach. We compare the number of transformations (T2M, M2M, and M2T) to be defined with our approach and using direct imports/exports between the tools.
- **RQ3:** *How feasible and applicable is the proposed approach in practice?* **Rationale:** We define the two refactorings needed to close the gap between two abstract syntax languages ($\mathcal{L}_{CardinalityBasedFMs}$ and $\mathcal{L}_{RelaxedFMs}$). We provide two different implementations of the refactorings: using Henshin Arendt et al. (2010) and Java. The objective is not to maintain both implementations, but to analyze the viability and applicability of the approach in practice and to select the best implementation for further transformations.
- **RQ4:** *Are the implemented refactorings correct, complete, and scalable?* **Rationale:** Refactorings should maintain the semantics of feature models (soundness), representing the same set of products with a different set of language constructs (completeness) even when dealing with large-scale feature models (scalability). We evaluate and compare these properties in the two implementations to decide which one can be a good candidate to develop further transformations between language constructs.
- **RQ5:** *What specific language constructs should be considered in the definition of a feature modeling language?* **Rationale:** The alternative approach to $\mathbb{CAF}$ is the definition of a completely new feature modeling language that supports all the requirements of the current domains (Horcas et al., 2022). Our approach helps to understand which language constructs should be provided in the definition of such a language in order to provide enough expressiveness to represent the requirements of current domains such as edge computing.

6.1. Open-source implementation of $\mathbb{CAF}$

We provide an implementation of the abstract syntax[4] by using the Eclipse Modeling Framework (EMF) and Ecore metamodels (Steinberg et al., 2008). We choose EMF/Ecore because it is a well-known and stable MDE framework providing good support for multiple facilities such as model transformations, code generation, serialization of structured data models, and models@runtime. In addition, there exist several implementations of EMF/Ecore in different languages beyond EMF-Java and Eclipse, and thus, Ecore metamodels can be directly used in Python or C++ by using the PyEcore[5] and EMF4CPP[6] frameworks respectively.

EMF/Ecore provides mechanisms that directly implement the concepts used to formalize our abstract syntax (Section 4). It provides *classes* to represent entities, *attributes* to represent properties of entities, and *relations* to represent relationships between

4 The implementation is available in https://github.com/CAOSD-group/rhea.
5 https://github.com/pyecore/pyecore
6 https://github.com/catedrasaes-umu/emf4cpp

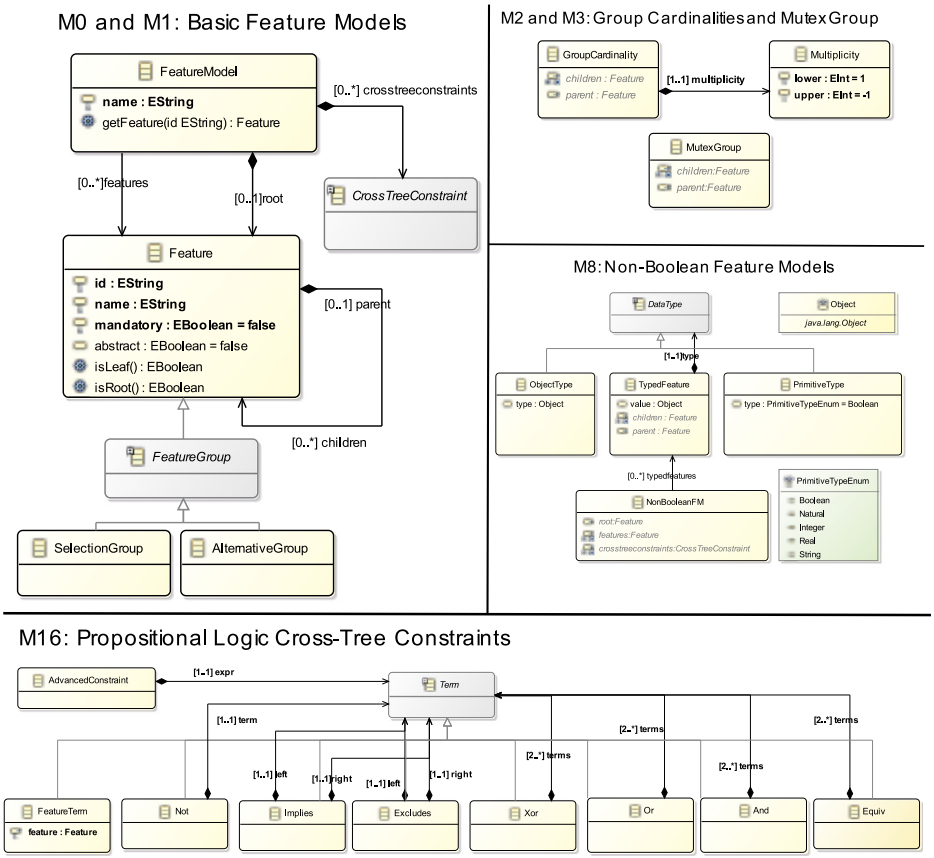


Fig. 8. Implementation $\mathbb{CAF}$ using EMF/Ecore metamodels (excerpt with metamodels M0, M1, M2, M3, M8, and M16).

the entities. The latter includes *Super Type* relations supporting the extends relation, and *compositions* and *(bi-directional) references* that support the composition relation. The dependency relation between metamodels is implemented in EMF/Ecore as a referenced resource, making available all constructs defined in the referenced metamodel.

Fig. 8 illustrates how to implement the abstract syntax with EMF/Ecore. Metamodels M_0 and M_1 implement the most basic language construct for feature models. The feature model construct 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 and M_1 specifies the most generic type of feature model, while the `NonBooleanFM` class, in metamodel M_8, model specializations of the feature model by extending the `FeatureModel` class. The extension relation among different metamodels is explicitly shown by showing the attributes of the super type in italic and light font as for example in the `NonBooleanFM` class. The rest of language constructs 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 `Optional Feature`, `Mandatory Feature`, `Abstract 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. The criteria to decide whether a language construct is modeled as a class, attribute, or relationship is an implementation decision of its abstract syntax but independent from the semantics meaning of the language construct. Finally, the extension and composition relations between language constructs are implemented using the *Super Type* relation and the *composition* reference, respectively. For example, the `FeatureModel` class in M_0 exposes the `CrossTreeConstraint` class using the composition relation. The latter class is abstract and enables specifying concrete constraints in separate metamodels as in M_{16} where `AdvancedConstraints` extends `CrossTreeConstraint`, using the *Super Type* relation. Note that, in this case, the *Super Type* relation between language constructs of different metamodels is not explicitly shown in diagrams, but in the .xml source file of the metamodel. The same design is used in metamodels M_8 to define generic concepts for non-Boolean feature models and typed features, which allows specializing those concepts for other types of features like numerical features.

6.2. Results and discussion

In this section, we present the results of our evaluation by answering each research question. We also discuss the threats to validity associated with each experiment and research question.

RQ1: How expressive are the feature modeling languages of the existing tools?. We show how existing feature modeling languages used in the SPL tools discussed in Section 2.2 are modeled in $\mathbb{CAF}$, so that their expressiveness can be compared in terms of the expressiveness of the common language constructs. Table 8 maps each tool language with the abstract syntax $\mathcal{L}$ it supports and the level of expressiveness (illustrated in Fig. 9) with respect to

Table 8
Mapping between tool languages and abstract syntax in $\mathbb{CAF}$.

$\mathbb{CAF}$ representation of existing tools	Expressiveness
$\mathcal{L}_{SPLOT} = \bigcup M_0, M_{15-16}$	$E(\mathcal{L}_{SPLOT}) = E(\mathcal{L}_{RelaxedFMs})$
$\mathcal{L}_{FeatureIDE} = \bigcup M_{0-1}, M_{16}$	$E(\mathcal{L}_{FeatureIDE}) = E(\mathcal{L}_{RelaxedFMs})$
$\mathcal{L}_{Glencoe} = \bigcup M_0, M_{3-4}, M_{15-16}$	$E(\mathcal{L}_{Glencoe}) = E(\mathcal{L}_{RelaxedFMs})$
$\mathcal{L}_{FaMa} = \bigcup M_0, M_{3-4}, M_8, M_{10}, M_{15-16}$	$E(\mathcal{L}_{FaMa}) = E(\mathcal{L}_{RelaxedFMs})$
$\mathcal{L}_{PureVariants} = \bigcup M_{0-1}, M_3, M_{7-8}, M_{10}, M_{15-19}$	$E(\mathcal{L}_{PureVariants}) \subseteq E(\mathcal{L}_{CardinalityFMs})$
$\mathcal{L}_{Clafer} = \bigcup M_{0-4}, M_{7-9}, M_{12}, M_{14-19}$	$E(\mathcal{L}_{Clafer}) \subseteq E(\mathcal{L}_{NumericalFMs}) \bigcup E(\mathcal{L}_{CardinalityFMs})$

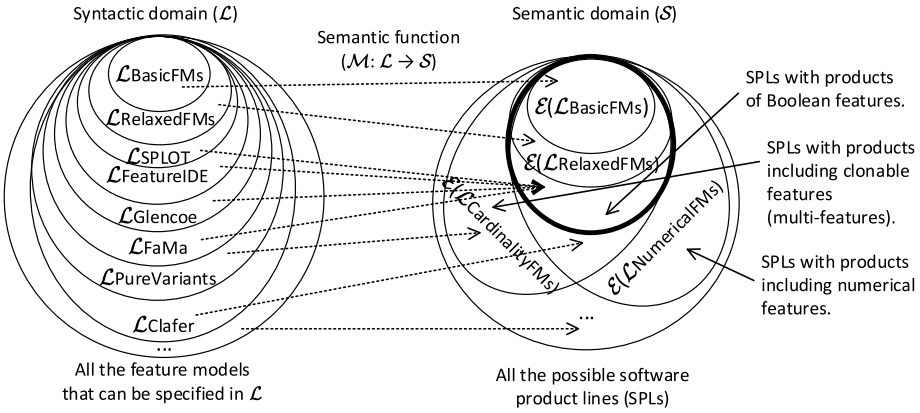


Fig. 9. Expressiveness of the feature modeling language of the existing tools.

the languages formalized in Section 4.3. On the one hand, all languages have at least the same expressiveness as $\mathcal{L}_{RelaxedFMs}$. That means that all languages can represent SPLs whose products are based on Boolean features expressed as arbitrary propositional logic formulae (*i.e.,* using the logical operators $\neg$, $\vee$, $\wedge$, $\Rightarrow$, and $\Leftrightarrow$). However, the language constructs used for each language sometimes is different. For instance, while $\mathcal{L}_{FeatureIDE}$ allows using abstract features (M_1) and arbitrary propositional formulae (M_{16}), $\mathcal{L}_{SPLOT}$ supports basic constraints (M_{15}) and propositional formulae expressed as clauses (*i.e.,* only using the $\neg$ and $\vee$ operators from M_{16}). $\mathcal{L}_{Glencoe}$ and $\mathcal{L}_{FaMa}$ also support the group cardinality construct from cardinality-based feature models (M_3 and M_4), but not the clonable feature construct (M_7). On the other hand, $\mathcal{L}_{PureVariants}$ and $\mathcal{L}_{Clafer}$ can be considered more expressive because they support clonable features (M_7), but it has not been formally demonstrated that clonable features can be represented with simpler language constructs such as those in $\mathcal{L}_{RelaxedFMs}$ due to the existence of advanced constraints (M_{17} and M_{20}). The same happens with $\mathcal{L}_{Clafer}$ and the numerical feature construct and its dependencies (M_8, M_9, M_{18}, and M_{19}), where there is no demonstrated evidence that those complex product lines with numerical features can be represented with simpler constructs, even deteriorating other properties of the feature models (*e.g.,* structure or size). A detailed study of these concerns is out of the scope of this paper, but we present the basis for dealing with them from a common point of view based on the language constructs.

Conclusion: *All feature modeling languages of the analyzed tools allow specifying SPLs whose products are based on Boolean features. Non-Boolean SPLs require more expressive language constructs such as numerical features, which are currently only supported in $\mathcal{L}_{Clafer}$. We have shown how the use of CAF allows reasoning about the expressiveness of these and other feature modeling languages.*

Threats to validity for RQ1:

Quantitative assessment of expressiveness (construct validity). We have compared the expressiveness of the feature modeling language of existing tools by mapping them to well-known languages/dialects in feature modeling: $\mathcal{L}_{BasicFMs}$ (Kang et al., 1990; Schobbens et al., 2007), $\mathcal{L}_{RelaxedFMs}$ (Knüppel et al., 2017), $\mathcal{L}_{CardinalityFMs}$ (Czarnecki et al., 2005), and $\mathcal{L}_{NumericalFMs}$ (Alférez et al., 2019; Munoz et al., 2019). We have evaluated expressiveness considering the language constructs they offer, whose semantics have already been formalized (Schobbens et al., 2007) and their expressiveness evaluated Knüppel et al. (2017) in the literature. A more appropriate solution to measure the expressiveness of languages is to determine the percentage of SPLs that can be represented with each abstract syntax language, as demonstrated by Knüppel et al. (2017). However, this procedure is only applicable in practice for a very small number of features (*i.e.,* a maximum of four or five features), as the generation of all possible feature models with a specific language $\mathcal{L}$ and their SPLs is an NP-hard problem (Knüppel et al., 2017).

Application beyond feature modeling (external validity). In this paper, we have focused on feature modeling. The interoperability between feature models and other variability approaches (decision models (Schmid et al., 2011), OVM (Pohl et al., 2005)) have been addressed in other related studies such as the TRAVART approach (Feichtinger et al., 2021) and such interoperability is out of scope of our paper. However, the concepts and formalization of our common and modular abstract syntax may be extended to support other variability modeling approaches. For instance, in another realization of $\mathbb{CAF}$, the modular metamodels may use "decisions" as the first citizen concept (unit of variability) instead of "features". However, a realization of $\mathbb{CAF}$ for other approaches (*e.g.,* decision models) requires identifying the language constructs of such approaches and defining the common abstract syntax of their modeling concepts. Then, mapping the existing languages (*e.g.,* existing concrete syntaxes of decision models) to

Table 9
Metamodels needed to represent feature modeling languages of existing tools, and transformations required to provide interoperability between the tools at the language construct level. We highlight in gray those metamodels for which a refactoring maintaining the semantics of the feature models can be defined in order to provide interoperability among tools.

Language	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}	M_{15}	M_{16}	M_{17}	M_{18}	M_{19}	M_{20}	M_{21}	M_{22}
$\mathcal{L}_{SPLOT}$	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●	○	○	○	○	○	○
$\mathcal{L}_{FeatureIDE}$	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○
$\mathcal{L}_{Glencoe}$	●	○	○	●	●	○	○	○	○	○	○	○	○	○	○	●	●	○	○	○	○	○	○
$\mathcal{L}_{FaMa}$	●	○	○	●	●	○	○	○	●	○	●	○	○	○	○	●	●	○	○	○	○	○	○
$\mathcal{L}_{PureVariants}$	●	●	○	●	○	○	○	●	●	○	●	○	○	○	○	●	●	●	●	●	○	○	○
$\mathcal{L}_{Clafer}$	●	●	●	●	●	○	○	●	●	●	○	○	●	○	●	●	●	●	●	●	○	○	○
M2M transf.:	–	LC	R	R	R	–	–	S	LC	S	LC	–	LC	–	LC	G	-	LC	LC	LC	–	–	–

●: Metamodel needed. ○: Metamodel not needed.
R: Refactoring from Fig. 6 needed. S: Specialization needed. G: Generalization needed. LC: No transformation applies, and a new language construct need to be defined. -: Non Applicable because all tools needed this metamodel or none tool needed it.

the new realization of $\mathbb{CAF}$ and defining the appropriate model transformations.

RQ2: What is the complexity of providing interoperability between existing feature modeling languages? Once the existing feature modeling languages have been represented in $\mathbb{CAF}$, we can use the following interoperability scenario to close the gap between the expressiveness and tool support for managing and analyzing feature models: Given a feature model in a concrete feature modeling language, we represent the feature model in our common abstract syntax (using a parser), and then apply our refactorings to the feature model to obtain another feature model without the specific language constructs considered. Finally, we translate the resulting feature model into a concrete language (using a writer). We measure the complexity of our approach by counting the number of transformations (*i.e.,* T2M, M2M, and M2T) that must be defined using our approach and without it (*i.e.,* defining direct imports/exports functionalities) to provide interoperability between the tools considered. Using imports/exports, we will need to define two transformations (if possible) between each pair of languages, requiring a total of 30 transformations to provide full interoperability among the six tools considered. Taking into account the seven already existing imports/exports between tools (Table 3 from Section 2.2), 23 imports/exports still need to be defined. Within our approach, we need to define a parser (T2M) and a generator (M2T) for each tool to represent it in $\mathbb{CAF}$, requiring a total of 12 transformations plus the required M2M transformations between the language constructs in $\mathbb{CAF}$. Table 9 summarizes the M2M transformations that are required to provide interoperability between the tools at the language constructs level. We can observe that not all cases require a transformation, such as the case where all tool languages use a metamodel (*e.g.,* M_0, M_{16}) or no tool language uses it (*e.g.,* M_5, M_6). There are other cases in which a transformation does not exist and we need to define an explicit language construct to represent the modeling concept, such as in the case of abstract features (M_1) or features with attributes (M_{10}). For those language constructs that there is an M2M transformation, in order to provide the minimum interoperability between the six tools, we need to provide, at least, those refactorings that have been formalized and guarantee that the semantics of the feature models are maintained (highlighted in gray in Table 9). These refactorings correspond to the language constructs of the $\mathcal{L}_{CardinalityBasedFMs}$ language that can be translated to $\mathcal{L}_{RelaxedFMs}$ language. Concretely, the mutex group construct (M_2) which is part of $\mathcal{L}_{Clafer}$, but not in other languages, and the group cardinality construct (M_3 and M_4) which is lacking in $\mathcal{L}_{SPLOT}$, $\mathcal{L}_{FeatureIDE}$, and $\mathcal{L}_{PureVariants}$. As a result, with our approach, we need to define up to 14 transformations (6 T2M, 2 M2M and 6 M2T), in contrast to 30 imports/exports transformations (73% of fewer transformations). Interoperability between languages with different expressiveness (*e.g.,*$\mathcal{L}_{NumericalFMs}$ and $\mathcal{L}_{RelaxedFMs}$) will imply some loss of information in the feature model. For example, the specialization needed to support numerical features (M_9) in other languages. Other transformations such as the generalization of simple constraints (M_{15}) to propositional logic constraints (M_{16}), in the case of $\mathcal{L}_{FeatureIDE}$ is straightforward (Batory, 2005).

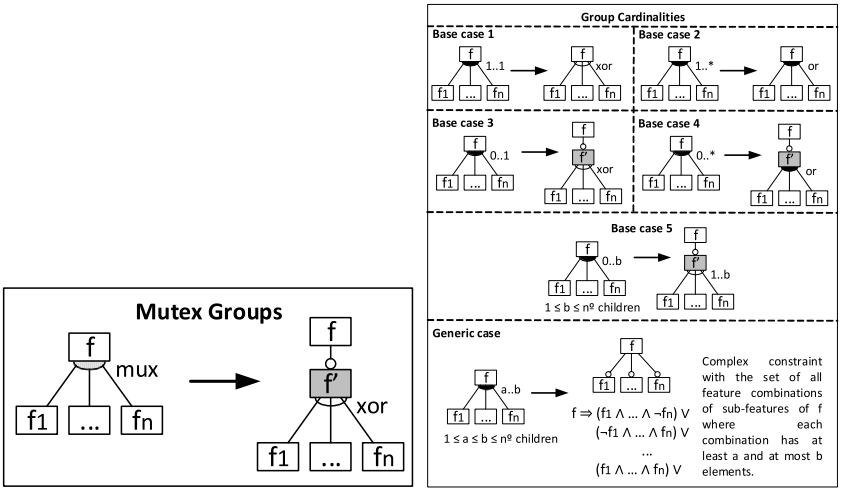
Conclusion: *The complexity of providing interoperability is given by the complexity of defining the required model transformations, which is reduced when defining them at the language construct level using a common metamodel as a central piece for the transformation process.*

Threats to validity for RQ2:

Complexity of model transformations (construct validity). We have measured the complexity of our approach by counting the number of transformations that need to be defined compared to the number of imports/exports needed. We have also assumed that the complexity of defining the model transformations is the same as for the imports/exports. In fact, defining an import/export functionality requires us to deal with all language constructs of the source/target language in each import/export, while with our approach we only need to deal with the entire set of language constructs once for each language in order to represent it in $\mathbb{CAF}$, and then the developer can focus on the model transformation of a specific language construct in $\mathbb{CAF}$ regardless of its concrete syntax. A more precise quantitative assessment of the complexity of the model transformations is part of our future work.

RQ3: How feasible and applicable is the proposed approach in practice?. To demonstrate the applicability of the approach, we provide an implementation of the two refactorings required to close the gap between the abstract syntax of the $\mathcal{L}_{CardinalityBasedFMs}$ and $\mathcal{L}_{RelaxedFMs}$ languages. Concretely, we define the refactorings to map the mutex group construct (M_2) to basic feature models (M_0 and M_1), and to map the group cardinality construct (M_3) to basic feature models with propositional logic cross-tree constraints (M_0, M_1, and M_{16}) as illustrated in Fig. 10.

- **Refactoring for mutex group** (Fig. 10(a)). Mutex groups are a kind of decomposition relations that often occurs in the software systems domain (Berger et al., 2013) (*e.g.,* in KConfig and CDL systems). A mutex group specifies a group of features where at most one feature can be selected, that is equivalent to a group cardinality with multiplicity 0..1. The refactoring for mutex groups have been already formalized in Knüppel et al. (2017) as follows: if a feature f is a mutex group decomposed into feature $f1, \ldots, f_n$, we change f's decomposition type to a single feature with one optional abstract sub-feature f'. Feature f' becomes an alternative (xor) group with subfeatures $f1, \ldots, f_n$.



(a) Refactoring for *mutex groups*. (b) Refactoring for *group cardinalities*.

Fig. 10. Mapping between language constructs.

- **Refactoring for group cardinalities** (Fig. 10(b)). Group cardinalities are feature groups with a custom multiplicity specifying a lower (a) and a upper (b) bound where at least a children and at most b children must be selected in a valid configuration. Examples of languages (*e.g.,* Clafer) supporting custom group cardinalities can be found in Czarnecki et al. (2005). The refactoring for group cardinalities have been also formalized in Knüppel et al. (2017) as follows: if a feature f is a group cardinality with multiplicity $< a..b >$, we change f's decomposition type to a single feature with n optional features being n the number of sub-features $(f1, \ldots, f_n)$ of f and add the following propositional logic constraint:

$$f \Rightarrow \bigvee_{M \in P_{a,b}} \left(\bigwedge_{f_i \in M} f_i \wedge \bigwedge_{f_j \in \{f' | f' \text{ is child of } f\} \setminus M} \neg f_j \right) \quad (1)$$

with $P_{a,b} = \{A \in 2^{\{f' | f' \text{ is child of } f\}} | a \leq |A| \leq b\}$ being the set of all feature combinations of sub-features of f where each combination has at least a and at most b elements (see generic case in Fig. 10(b)). In order to simplify the resulting/refactored feature model, we identify a set of base cases where the group cardinality can be translated to a basic language construct (*i.e.,* selection group or alternative group), according to its multiplicity. Base cases from 1 to 5 in Fig. 10(b) illustrate those scenarios where we obtain simpler feature models in terms of language constructs without the need of introducing complex (propositional logic) constraints. Note that base case 5 follows the same refactoring as for mutex group, requiring then the application of the generic case.

Both refactorings have been implemented using two different technologies to demonstrate their viability and select the most appropriate one in order to implement further refactorings: (1) using the Henshin transformation language (Arendt et al., 2010) for a pure model-driven development approach and achieving a higher level of abstraction for the transformations; and (2) using object-oriented programming by accessing directly with a programming language to the classes and relations defined in the metamodels which allows using our approach outside Java and the EMF/Ecore Eclipse environment (*e.g.,* in Python or C++ using the PyEcore or EMF4CPP, respectively).

Refactorings as Henshin model transformations. Henshin Arendt et al. (2010) is a model transformation language based on algebraic graph transformations. We choose Henshin to implement the model transformations because of its several key benefits in contrast to other transformation languages (*e.g.,* ATL): (i) changes to a given input model (in our case, a feature model) are expressed using graphical rules facilitating the inspection of the rules by a visual syntax; (ii) those rules support a largely declarative specification of transformations described below; (iii) Henshin concepts to specify rules aligns well with the concepts used in the EMF/Ecore implementation (*i.e.,* nodes, edges, attributes); and (iv) Henshin provides an execution engine which applies the rules to the input model and can be executed programmatically using its API.

A *Henshin model transformation* consists of a set of transformations *units*. There are two kind of units: (1) *rules* that are the basic building blocks for model transformations, and (2) *composite units* which enable the orchestration of multiple rules in a control flow. A rule comprises two graphs: the *Left-hand side (LHS) graph* that describes a pattern to be matched in the input model; and the *Right-hand side (RHS) graph* that specifies a change of the input model. The graphical editor merges LHS and RHS into an integrated representation. A graph specifies model patterns on the abstract syntax level by means of *nodes*, *edges*, and *attributes*, similar to the concepts of EMF/Ecore (classes, relations, and attributes). Nodes refer to objects (*e.g.,* features, constraints, feature models, in our case); edges refer to references between objects (*e.g.,* relationships between features); and attributes inside nodes refer to properties (*e.g.,* feature's names). A transformation rule can be applied whenever the LHS graph is matched by a given feature model instance. A *matching* is an assignment of all variables (nodes, edges, and attributes) occurring in LHS to concrete values. Elements in the LHS graph are matched with the input model and maintained (`<<preserved>>`) or removed (`<<delete>>`), while elements in the RHS pattern are created (`<<create>>`) as for example to add a new feature in the model. Additional restrictions can be imposed to prevent the existence of elements (`<<forbid>>`) as for example to avoid duplicate features. Any element not mentioned in both LHS/RHS remains unchanged, so transformations only need to define the differences between the models. A transformation can also accept parameters such as the name of a feature. Finally, composite units specify control flow and have a fixed number of sub-units, allowing for arbitrary nesting. Several types of units are available such as loop units

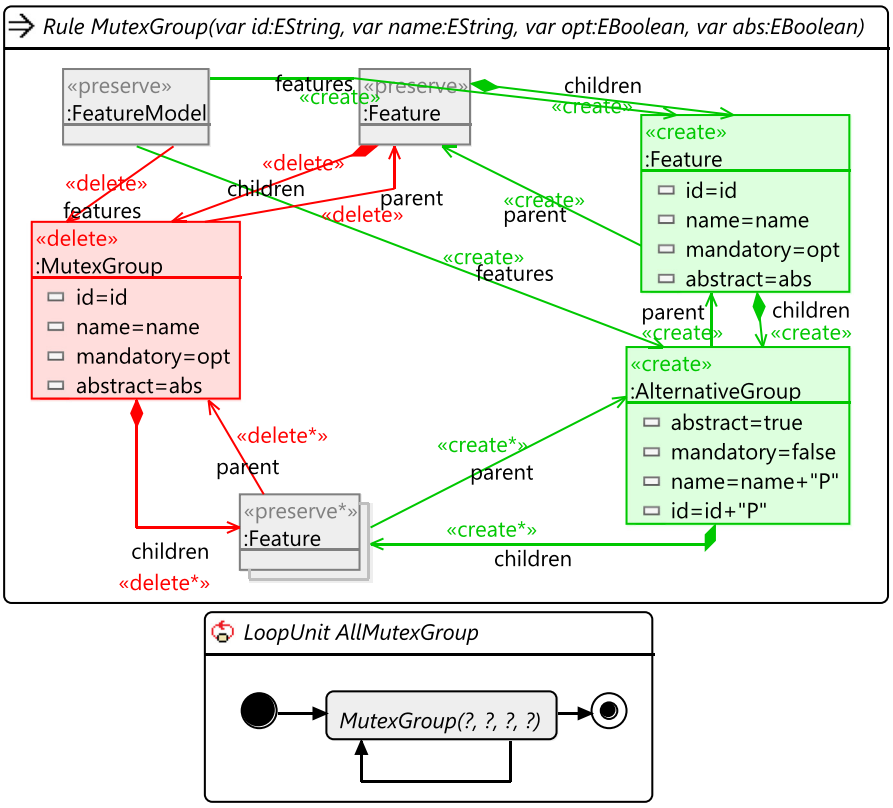


Fig. 11. Henshin transformation for mutex groups.

(to execute a particular rule as often as possible), iterate units (to execute a rule a fixed number of times), sequential units or conditional units, among others.

- **Mutex group refactoring.** We implement the mutex group refactoring as a single rule (see Fig. 11) that translates the mutex group language construct (the `<<delete>>` `MutexGroup` node) for a combination of a single feature (`<<create>>` `Feature` node) and an optional alternative group (`<<create>>` `AlternativeGroup` node), and we also update the edges for the parent and the children of the corresponding nodes. Then, to guarantee completeness, we add an additional loop unit (`AllMutexGroup`) that encapsulates the `MutexGroup` rule in order to execute the refactoring as often as possible to translate (or refactor) all occurrences of the same language construct in a given input model. Note that we also need an additional Henshin rule, similar to the described one, but that matches with the root feature in case that the root of the feature model was a mutex group.
- **Group cardinalities refactoring.** The group cardinalities refactorings are implemented as follows. Each base case of the group cardinalities is implemented as a Henshin rule and a loop unit to guarantee completeness similarly to the mutex group refactoring presented in Fig. 11. Additionally, we add a sequential unit which call every loop unit to apply all base cases sequentially in the input feature model. The generic case adds a complication due to we need to generate a cross-tree constraint with all feature combinations of sub-features of f taken k positive feature terms and $n - k$ negative feature terms, for all k from a to b. Fig. 12 shows the Henshin rule that generates those combinations for $k = 2$. We first generate the skeleton of the constraint (`CreateConstraint` rule). Second we generate all possible combinations of the positive terms sub-features of f for all k values. In Fig. 12 we show the transformation for $k = 2$ (`AddPositives_F_k2` and `AddPositive_F_K2` rules). Since k depends on the lower and upper multiplicity of a particular instance of the group cardinality construct, and the `AddPositive` rule differs for different values of k, we programmatically generate the `AddPositive` rule for every k of the group cardinality considered. Third, we add the negative terms sub-features of f (`AddNegatives` and `AddNegative` rules). Note that we need to use `Trace` nodes to keep track of the elements already considered in the transformation, and then we remove those traces (`CleanFeatureTrace` rule). Finally, we transform the group cardinality language construct to a selection group construct using a similar rule as for the base cases and the mutex group (see Fig. 11). The complete refactoring transformation as well as the code to generate the corresponding Henshin rules for the feature combinations can be found online.[7]

Refactorings as Java transformations. The implementation of the abstract syntax with the EMF/Ecore framework allows us working directly with the classes, attributes, and methods/relations defined in the metamodels. Thus, we can also implement model transformations between language constructs using a traditional

[7] https://github.com/CAOSD-group/rhea

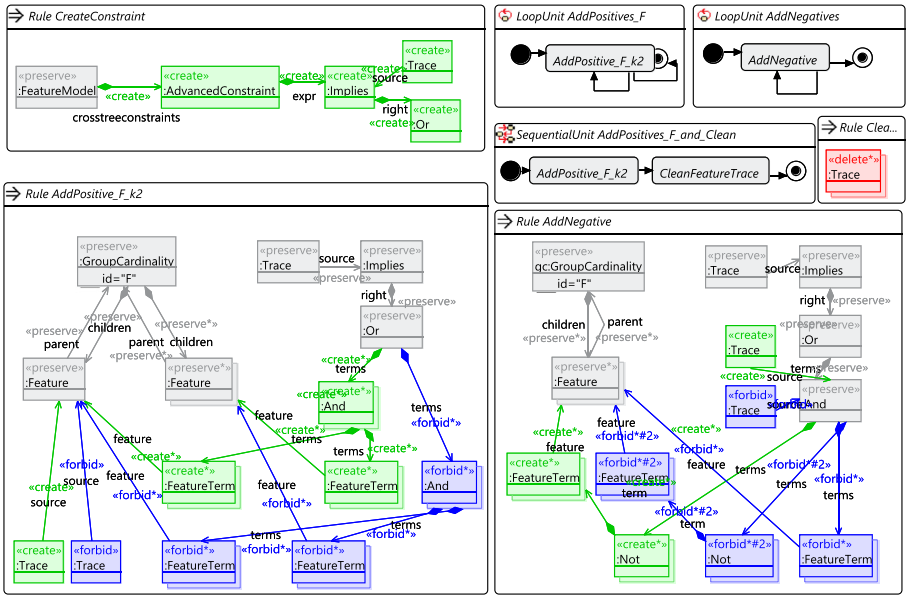


Fig. 12. Generation of the propositional logic constraints in the group cardinality refactoring.

programming paradigm such as object-oriented programming in Java. Algorithm 1 illustrates the code for the mutex group refactoring using the concepts modeled in our metamodels. The algorithm finds and transforms every instance of the language construct (*i.e.*, mutex group in this case). For each language construct to be translated, the algorithm creates the objects of the target language construct(s), copies the attributes' values (*e.g.*, id, name, properties), and updates their relationships (*e.g.*, parents and children features). The code for the group cardinality refactoring (Algorithm 2) follows the same structure with the addition of generating the cross-tree constraints for all possible combinations of the feature group, similarly as explained in the Henshin rule. The complete Java implementation of the refactorings can be found online in the code repository.

Algorithm 1 Mutex group refactoring.

Input: feature model (*fm*)
Output: refactored feature model (*fm*)
1: **procedure** MutexGroupRefactoring(*fm*)
2: **for** *f* ∈ *fm*.features **do**
3: **if** *f* `instanceof` MutexGroup **then**
▷ Create the new feature, copy properties, and update relations
4: *feature* = new Feature(id=*f*.id, name=*f*.name, mandatory=*f*.mandatory, abstract=*f*.abstract)
5: **if** ¬*feature*.isRoot() **then**
6: *feature*.parent = *f*.parent
7: *f*.parent.children.remove(*f*)
8: *feature*.parent.children.add(*feature*)
9: **end if**
▷ Create the abstract alternative group and update relations
10: *group* = new AlternativeGroup(id=*f*.id+"P", name=*f*.name+"P", mandatory=**false**, abstract=**true**)
11: *group*.parent = *feature*
12: *feature*.children.add(*group*)
13: **for** *child* ∈ *f*.*children* **do**
14: *child*.parent = *feature*
15: **end for**
▷ Update model's features
16: *fm*.features.remove(*f*)
17: *fm*.features.add(*feature*)
18: *fm*.features.add(*group*)
19: **end if**
20: **end for**
21: **return** *fm*
22: **end procedure**

Algorithm 2 Group cardinality refactoring.

Input: feature model (*fm*)
Output: refactored feature model (*fm*)
1: **procedure** GroupCardinalityRefactoring(*fm*)
2: **for** *f* ∈ *fm*.features **do**
3: **if** *f* `instanceof` GroupCardinality **then**
▷ Convert to a normal feature, copy properties, and update relations
4: *feature* = new Feature(id=*f*.id, name=*f*.name, mandatory=*f*.mandatory, abstract=*f*.abstract)
5: **if** ¬*feature*.isRoot() **then**
6: *feature*.parent = *f*.parent
7: *f*.parent.children.remove(*f*)
8: *feature*.parent.children.add(*feature*)
9: **end if**
▷ Convert children to optional features
10: **for** *child* ∈ *f*.children **do**
11: *child*.mandatory = **false**
12: *child*.parent = *feature*
13: *feature*.children.add(*child*)
14: **end for**
▷ Generate cross-tree constraint
15: *ctc* = new AdvancedConstraint()
16: *ctc*.expr = new Implies()
17: *ctc*.expr.left = new FeatureTerm(feature=*feature*)
18: *ctc*.expr.right = new Or()
19: *n* = *feature*.children.size()
20: **for** *k* ∈ [*f*.multiplicity.lower, *f*.multiplicity.upper] **do**
21: **for** $p \in \binom{n}{k}$ **do**
22: *positiveTerms* = *feature*.children.getAll(indexes=*p*)
23: *negativeTerms* = *feature*.children \ *positiveTerms*
24: *andOp* = new And()
25: **for** *posT* ∈ *positiveTerms* **do**
26: *andOp*.terms.append(new FeatureTerm(feature=*posT*))
27: **end for**
28: **for** *negT* ∈ *negativeTerms* **do**
29: *andOp*.terms.append(new Not(new FeatureTerm(feature=*negT*)))
30: **end for**
31: *ctc*.expr.right.terms.append(*andOp*)
32: **end for**
33: **end for**
▷ Update model's features
34: *fm*.features.remove(*f*)
35: *fm*.features.add(*feature*)
36: *fm*.features.add(*group*)
37: **end if**
38: **end for**
39: **return** *fm*
40: **end procedure**

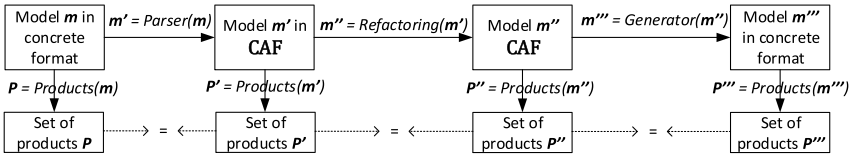


Fig. 13. Soundness validation of our interoperability approach.

Conclusion: *Defining the refactorings at the language construct level facilitates the analysis and comprehension of the transformations. Transformations can be implemented with model-driven engineering technologies or object-oriented programming using the same metamodel, which makes our approach more applicable in practice.*

Threats to validity for RQ3:

Technical debt in model-driven technologies (internal validity). Another threat to validity is related to the possible errors in the experimental materials and tools used. Model-driven engineering tools are continuously evolving and the related technology suffers from a high technical debt. To mitigate this threat, we have relied on well-known tools for model-driven development like the EMF/Ecore framework (Steinberg et al., 2008) and the Henshin transformation language (Arendt et al., 2010), which are under active development and are also well documented.[8]

Refactorings for other feature modeling languages (external validity). We have evaluated only two types of refactorings (mutex group and group cardinality) related with the $\mathcal{L}_{CardinalityFMs}$ language. First, we used those refactorings because they have already been formalized (Knüppel et al., 2017) and demonstrated to have the same expressiveness as $\mathcal{L}_{RelaxedFMs}$. Other refactorings, such as clonable features and numerical features, have not yet been formalized and can be seen as open challenges in the feature modeling community. Second, these are the only language constructs that are missing in the existing feature modeling tools to provide full interoperability between them, while maintaining the same expressiveness (*i.e.,* the expressiveness of the $\mathcal{L}_{RelaxedFMs}$ language). Other refactorings for $\mathcal{L}_{RelaxedFMs}$ are not needed in the tools analyzed because the respective language constructs are available in all tools or in none tool (see Table 9). Finally, we used those refactorings to illustrate our approach, since the objective of this article is not to provide a refactoring for each existing language construct, but to demonstrate the viability of an interoperability approach like the one presented in this paper. The implementation of further refactorings as well as specializations are in our planning agenda.

RQ4: Are the implemented refactorings correct, complete, and scalable?. We quantitatively evaluate the refactorings implemented using metrics to show the soundness (correctness), completeness, and scalability. We use the interoperability scenario illustrated in Fig. 13 in which we focus on ensuring that the refactorings for the language constructs (*i.e.,* mutex groups and group cardinalities) are correct. The evaluation corpus includes two sets of feature models:

- **Base case testing feature models.** A set of 56 testing feature models (8 for the mutex-group and 48 for the group-cardinality language construct) manually built to consider every base case of the refactorings. The expected refactoring for each feature model is known before the application of the refactoring and all configurations and products of the feature model can be enumerated in order to compare them with those generated by the output refactored model. The size of these feature models varies from 3 to 30 features in order to be able to enumerate and compare all their configurations. We use this set of feature models to evaluate the soundness and the completeness of our refactorings.
- **Randomly-generated feature models.** A set of 224 feature models automatically generated to contain a specified large number of features (from 250 to 7000 features) and with a percentage of different types of language constructs (*e.g.,* group cardinalities, mutex groups, optional and mandatory features,…). We use this set of feature models to evaluate the scalability of our refactorings.

The experiments were carried out on a desktop computer with AMD Ryzen 5 2400G, 3.6 GHz, 16 GB of memory, and Windows 10 Pro. We use the Eclipse EMF/Ecore framework with Java 13, and Henshin 1.6.0.

Soundness of the interoperable scenario. As shown in Fig. 13, we want to demonstrate that given a feature model, the application of the different model transformations leads to a feature model that contains exactly the same products than the original one.

To demonstrate the soundness of our approach, we generate each valid product of the input feature model and of the output (transformed) feature model, and compare both set of products. We perform this checking in each transformation step of our approach to guarantee that the semantics of the feature model does not change during the pipeline. We use the Clafer tool[9] to enumerate the products. Since the product operation (Benavides et al., 2010) is costly or even infeasible for large-scale feature models, we use the pre-defined set of feature models manually generated (*i.e.,* the base case testing feature models) that covers all the casuistic of the refactorings and that their products can be enumerated efficiently (see Fig. 14). These feature models cover all base cases of the refactorings as well as all possible structural combinations of the language constructs to be refactored (*e.g.,* multiple nesting of the language constructs to be refactored), so that any other (large) feature model can fit into one of these base case models. For mutex-group we define 8 feature models, one for each base case presented in Fig. 14. For instance, BC5 corresponds to the case in which a child of a mutex-group is also a mutex-group, and thus, the refactorings should transform both instances; while BC8 represents the case in which the mutex-group is a child of any other kind of group like a xor-group or a selection group. For group-cardinalities, we define up to 48 feature models covering the 8 base cases defined in Fig. 14 for each of the 6 possibilities of the group-cardinality refactoring with regard to its multiplicity, which were presented in Fig. 10(b). In all cases, our refactorings (both in Henshin and Java) maintain the set of products of the feature models after applying each model transformation.

8 EMF/Ecore: https://www.eclipse.org/modeling/emf/, Henshin: https://www.eclipse.org/henshin/

9 https://www.clafer.org/

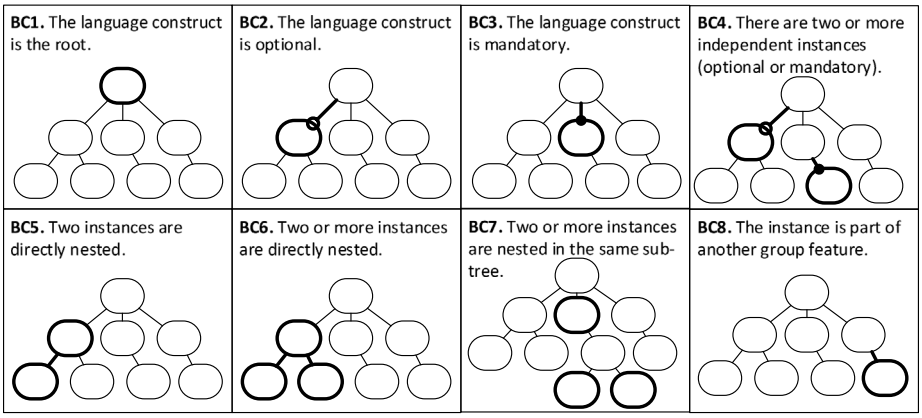


Fig. 14. Base cases used to validate the soundness of our approach.

Completeness of the refactorings. Given a feature model, applying a refactoring for a specific language constructs leads to a feature model which does not contain any instance of such language construct. That is, all instances of the language constructs to be refactored have been successfully transformed to the corresponding target language construct. To demonstrate the completeness of our refactorings we count the number of instances of a specific language construct (*e.g.,* group cardinality) to verify if no remaining instances are present in the transformed feature model. We apply this metric to the entire evaluation corpus of feature models. In all cases, the resulting feature models do not contain any instance of mutex-groups nor group cardinalities as they have been successfully refactored to the appropriate language constructs (in both the Henshin and Java implementations).

Scalability. Given a feature model, how long does the refactoring take to transform every instance of the language construct to the target construct? We use the execution time as a metric to evaluate the scalability of our refactorings. We have generated a set of random feature models with different number of features and different language constructs. We generate feature models which contain up to 7000 features, with a 10% to 20% of language constructs (mutex-group and group cardinalities) to be transformed. We use this ratio of language constructs in order to maintain the consistency of the feature model. A higher ratio will degrade the structure and naturalness of the feature model because of inconsistencies with the number of children that each feature group can have Schobbens et al. (2007) (*i.e.,* a feature group must have at least two children).

For each refactoring and feature model, we performed 30 runs, and calculated the medians, means and standard deviations for the execution time (see Table 10). The table shows the number of features before and after applying the refactorings since some of the refactorings can incorporate new features to the model (*e.g.,* mutex group always incorporates one extra feature). We also show the number of refactorings performed that corresponds with the number of instances of the language construct to be transformed. We can observe a significant difference between the performance of the pure MDD (Henshin) and the OOP (Java) implementations, where the Java refactorings outperform the Henshin versions in both type of language constructs. In fact, as illustrated in Figs. 15 and 16, Henshin refactorings show an exponential increase with the size of the feature model (around 7 s for a feature model of 7000 features and 1400 refactorings for mutex groups) while Java refactorings shows a polynomial increase (0.5 s for the same feature model). The difference is higher for the group cardinality language construct where Henshin takes more than 3 min to refactor 1400 groups in a feature model of 7000 features, while the Java version only takes around 0.5 s. We can conclude that from the point of view of the performance, refactorings in Java are superior to Henshin refactorings. However, Henshin refactorings allow following a complete MDD approach maintaining a higher degree of abstraction, facilitating the inspection of the rules by a visual syntax.

Conclusion: *Refactorings are correct since the transformed feature model has the same semantics as the input model (*i.e., *the same set of products). Refactorings are complete because the transformed feature model do not contain any instances of the refactored language construct. The scalability of the refactorings depends on the implementation which can be exponential due to the overhead introduced by MDD technologies regardless the inner complexity of a specific refactoring.*

Threats to validity for RQ4:

Experimentation set-up (internal validity). A threat to internal validity is our choice of feature models and their sizes for the evaluation corpus (*i.e.,* 56 models with up to 30 features) and the configuration parameters for the randomly-generated feature models (*i.e.,* 224 models with up to 7000 features and with 20% of a specific language construct). The reasons to use those values have been explained and discussed in RQ4. While a different set of models with other parameters will make the result of our experiments more confident, we believe that the chosen setup allows us to demonstrate the correctness, completeness, and scalability of the evaluated refactorings. While a detailed evaluation of the impact of each parameter (*e.g.,* number of features, number of groups, configurations, etc.) in the refactorings is out of the scope of this paper, we plan, as our future work, to study how other configuration parameter for feature models affect the refactorings, and to perform a sensitivity analysis to study the robustness of the refactorings with regard to configuration parameters.

Real-world feature models (external validity). There exist a vast set of real-world feature models that could be used to evaluate our approach. However, those feature models are usually presented in research papers as feature diagrams illustrating the variability of a real-world system, but they are not available in any feature model format due to the lack of tool support for all language constructs required by those systems (Horcas et al., 2022). For instance, even for the Clafer language which is the most expressive, there is a lack of models that include the most advanced language constructs since common tools cannot support them. In fact, the Clafer community has made publicly available up to 516 feature models in Clafer[10] that vary in size and complexity (including 341 feature models translated from the SPLOT[11]

[10] https://github.com/gsdlab/clafer/tree/master/test/positive
[11] http://www.splot-research.org/

Table 10
Evaluation of the refactorings. For each language construct and feature model we present the number of features before and after applying the refactorings, the number of refactorings (#Refacts) performed and the execution time of both implementations. We present the mean, median and standard deviation for 30 executions.

Language construct	Feature model size		#Refacts	MDD (Henshin)			OOP (Java)		
	#Features before	#Features after		Mean Time (s)	Median Time (s)	Std Time (s)	Median Time (s)	Median Time (s)	Std Time (s)
Mutex group	500	600	100	0.05	0.04	0.02	2.90e−3	2.88e−3	8.62e−5
	1000	1200	200	0.14	0.13	0.04	0.01	0.01	2.37e−3
	2000	2400	400	0.42	0.42	0.03	0.04	0.04	2.93e−4
	3000	3600	600	0.89	0.88	0.06	0.08	0.08	6.35e−4
	4000	4800	800	1.51	1.49	0.08	0.16	0.16	8.11e−3
	5000	6000	1000	3.04	2.96	0.44	0.26	0.26	6.76e−3
	6000	7200	1200	4.65	4.69	0.46	0.38	0.38	1.02e−2
	7000	8400	1400	7.41	7.19	1.73	0.51	0.51	7.79e−3
Group cardinality (generic case *a..b*)	500	500	100	0.55	0.54	0.03	4.82e−3	4.81e−3	2.81e−4
	1000	1000	200	2.10	2.06	0.10	0.02	0.02	1.17e−2
	2000	2000	400	8.63	8.60	0.15	0.04	0.04	3.83e−4
	3000	3000	600	23.01	23.02	0.36	0.09	0.09	1.31e−3
	4000	4000	800	42.08	41.73	1.14	0.16	0.16	1.32e−3
	5000	5000	1000	74.26	73.95	2.64	0.24	0.24	3.42e−3
	6000	6000	1200	124.47	124.32	8.34	0.36	0.36	1.48e−2
	7000	7000	1400	198.78	197.02	9.70	0.48	0.48	9.89e−3

repository). However, after checking all those models, we found that there is a lack of models using the mutex-group and group cardinality language constructs, and only a few include one or two instances of those language constructs. In fact, those models match with our models considered in the suit test cases we have used. The reason for a lack of models with those specific language constructs (mutex-group and group cardinality) is twofold. First, most of the realistic feature models are synthesized from existing real-world systems by the academia, but SPL researchers tend to simplify them (*e.g.,* considering only xor and or-groups) losing variability information. Second, with the exception of Clafer, the majority of tools to manage feature models only considers basic language constructs (*e.g.,* optional, mandatory, xor and or-groups) Horcas et al. (2022) and feature models are translated from those less-expressive tools to Clafer. For instance, all the feature models in the SPLOT repository are basic. This proves even more the necessity of having an approach such as the presented in this paper.

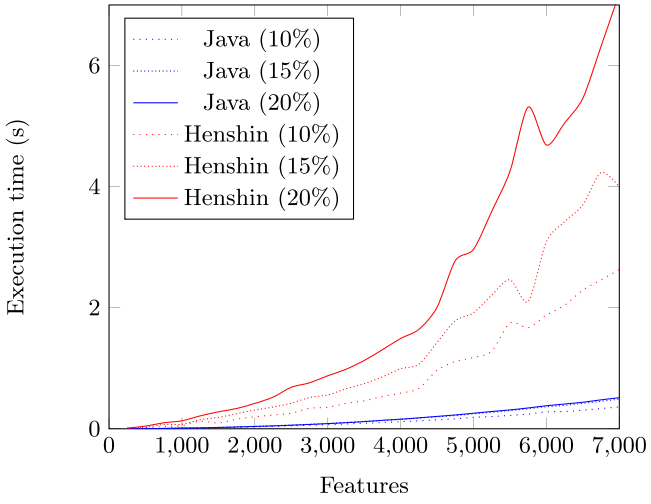


Fig. 15. Performance of the Mutex-group refactorings with respect to the size of the feature model (number of features) and percentage of Mutex-groups.

RQ5: What specific language constructs should be considered in the definition of a feature modeling language? Considering all M2M transformations, including refactorings, specialization, and generalizations, that need to be defined in order to provide full interoperability among existing feature modeling tools (see Table 9), an alternative to our approach is the definition of a completely new feature modeling language that supports all the language constructs already supported by existing tools (Horcas et al., 2022). Such a language should include those language constructs of $\mathbb{CAF}$ so that no transformations are needed to represent all SPL (see Fig. 5) without deteriorating the feature model structure (naturalness) or size (succinctness) (Schobbens et al., 2007). Our approach helps to identify those language constructs. Concretely, as exposed in Table 9, abstract features (M_1), mutex groups (M_2), group cardinalities (M_3 and M_4), and propositional logic constraints (M_{16}) should be exposed as explicit language constructs to ease the specification of feature models with the same expressiveness as $\mathcal{L}_{RelaxedFMs}$. A language construct for multi-features or clonable features (M_7) and for advanced constraints (M_{17}) should be exposed to achieve the full expressiveness of $\mathcal{L}_{CardinalityBasedFMs}$. Finally, to achieve the expressiveness of $\mathcal{L}_{NumericalFMs}$, numerical features ($M_9$), features with attributes (M_{10}), as well as constraints involving those features are needed (M_{18} and M_{19}).

Conclusion: *A simple language with only the language constructs that practitioners need, will be the best feature modeling language. However, the trade-off between a super set with all language constructs and a minimal set of language construct is something practitioners need to think about in the SPL community. Our approach allows feature models to be extended to support concepts that might be needed in specific domains.*

Threats to validity for RQ5:

Realization and implementation of $\mathbb{CAF}$ (conclusion validity). A different realization of $\mathbb{CAF}$ from the proposed in Section 4.2 may imply a different set of language constructs and thus different refactorings. The same occurs with a different implementation in the EMF/Ecore of $\mathbb{CAF}$ (Section 6.1) where the design decisions of the developers may imply a different set of refactorings between language constructs. For example, a developer can decide to model relationships between a feature and its parent using just cardinalities without the need of defining a language construct for each kind of feature group (Horcas et al., 2023). Our realization of $\mathbb{CAF}$ is based on existing language constructs from earlier work (Horcas et al., 2020) which have been demonstrated to be commonly used in practice in SPL (Horcas et al., 2022). Regarding

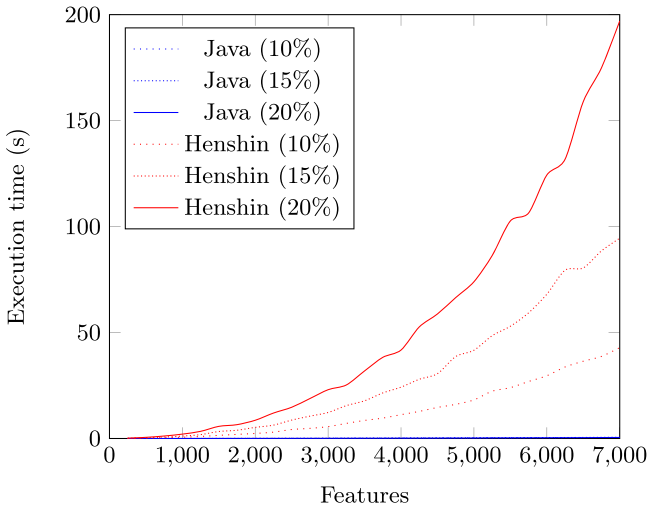


Fig. 16. Performance of the Group Cardinality refactorings with respect to the size of the feature model and percentage of group cardinalities.

the implementation, as a proof of concept that our approach is feasible in practice, we opted to try to explicitly represent each language construct, and thus, we define (if possible) a language construct for each modeling concept, while other feature modeling concepts have been defined as relationships between language constructs (*e.g.,* the *root* feature) or as properties of language constructs (*e.g., optional feature* as part of the *feature* construct).

7. Conclusions

We have presented an approach to provide interoperability of feature models based on their variability modeling constructs. Our approach relies on model-driven engineering techniques and advocates for a modular and extensible design that allows practitioners to decide which language levels support based on their needs. Independent-notation languages with different expressiveness levels can be generated as instances of the common abstract syntax by choosing only the language constructs that are needed to represent the same expressiveness that practical tools and existing languages offer. The abstract syntax of the language constructs enables the development of a set of reusable model transformations (*e.g.,* refactorings) to support better interoperability between existing languages and analysis capabilities that are not available for a full language with all possible constructs.

CRediT authorship contribution statement

Jose-Miguel Horcas: Investigation, Conceptualization, Software, Validation, Writing – review & editing. **Mónica Pinto:** Investigation, Methodology, Validation, Writing – review & editing. **Lidia Fuentes:** Work idea, Funding, Supervision, Resources.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Jose Miguel Horcas reports a relationship with University of Malaga that includes: employment and funding grants. Jose Miguel Horcas reports a relationship with University of Seville that includes: employment and funding grants. Monica Pinto reports a relationship with University of Malaga that includes: employment. Lidia Fuentes reports a relationship with University of Malaga that includes: employment. JMH has co-authored several papers with David Benavides, José A. Galindo, Rubén Heradio, and David Fernandez-Amoros this year. JMH has collaborated in a paper with Steffen Zschaler, Daniel Strüber, Alexandru Burdusel, and Jabier Martinez this year. MP and LF have co-authored a paper with Don Batory in the last three years. MP and LF have co-authored a paper with Dillian Gurov in the last three years. MP and LF have co-autohred a paper with Alessandro Vittorio Papadopoulos (Mälardalen University, Västerås, Sweden). MP, LF and JMH have co-authored a paper with Siobhán Clarke in the last years. MP, LF and JMH have participated in a national network of excellence in collaboration with the University of Seville, da Coruña, País Vasco, Castilla-La Mancha, Extremadura, Mondragón, Rey Juan Carlos, and Politécnica de Madrid.

Data availability

The implementation is available in a GitHub repository https://github.com/CAOSD-group/rhea.

References

Abele, A., Papadopoulos, Y., Servat, D., Törngren, M., Weber, M., 2010. The CVM framework - A prototype tool for compositional variability management. In: Fourth International Workshop on Variability Modelling of Software-Intensive Systems. In: ICB-Research Report, vol. 37, Universität Duisburg-Essen, pp. 101–105, URL http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf.

Acher, M., Collet, P., Lahire, P., France, R.B., 2013. FAMILIAR: A domain-specific language for large scale management of feature models. Sci. Comput. Program. (SCP) 78 (6), 657–681.

Al-Azzawi, A.F., 2018. Py?fml - A textual language for feature modeling. Int. J. Software Eng. Appl. (IJSEA) 9 (1), 41–53. http://dx.doi.org/10.5121/ijsea.2018.9104.

Alférez, M., Acher, M., Galindo, J.A., Baudry, B., Benavides, D., 2019. Modeling variability in the video domain: Language and experience report. Softw. Qual. J. 27 (1), 307–347. http://dx.doi.org/10.1007/s11219-017-9400-8.

Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., de Lucena, C.J.P., 2006. Refactoring product lines. In: 5th International Conference Generative Programming and Component Engineering. GPCE, ACM, pp. 201–210. http://dx.doi.org/10.1145/1173706.1173737.

Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G., 2010. Henshin: Advanced concepts and tools for in-place EMF model transformations. In: 13th International Conference on Model Driven Engineering Languages and Systems. MODELS, In: Lecture Notes in Computer Science, vol. 6394, Springer, pp. 121–135. http://dx.doi.org/10.1007/978-3-642-16145-2_9.

Asikainen, T., Männistö, T., 2009. Nivel: A metamodelling language with a formal semantics. Software Syst. Model. 8 (4), 521–549. http://dx.doi.org/10.1007/s10270-008-0103-2.

Asikainen, T., Männistö, T., Soininen, T., 2006. A unified conceptual foundation for feature modelling. In: 10th International Software Product Line Conference. SPLC 2006, IEEE Computer Society, pp. 31–40. http://dx.doi.org/10.1109/SPLINE.2006.1691575.

Atkinson, C., Kuhne, T., 2003. Model-driven development: A metamodeling foundation. IEEE Software 20 (5), 36–41.

Bashroush, R., Garba, M., Rabiser, R., Groher, I., Botterweck, G., 2017. CASE tool support for variability management in software product lines. ACM Comput. Surv. 50 (1), 14:1–14:45. http://dx.doi.org/10.1145/3034827.

Batory, D.S., 2005. Feature models, grammars, and propositional formulas. In: Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings. pp. 7–20. http://dx.doi.org/10.1007/11554844_3.

Benavides, D., 2019. Variability modelling and analysis during 30 years. In: From Software Engineering to Formal Methods and Tools, and Back. In: Lecture Notes in Computer Science, vol. 11865, Springer, pp. 365–373. http://dx.doi.org/10.1007/978-3-030-30985-5_21.

Benavides, D., Martín-Arroyo, P.T., Cortés, A.R., 2005. Automated reasoning on feature models. In: 17th International Conference on Advanced Information Systems Engineering. CAiSE, In: Lecture Notes in Computer Science, vol. 3520, Springer, pp. 491–503. http://dx.doi.org/10.1007/11431855_34.

Benavides, D., Rabiser, R., Batory, D.S., Acher, M., 2019. First international workshop on languages for modelling variability (MODEVAR). In: 23rd International Systems and Software Product Line Conference, Vol. A. SPLC, ACM, p. 46:1. http://dx.doi.org/10.1145/3336294.3342364.

Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35 (6), 615–636. http://dx.doi.org/10.1016/j.is.2010.01.001, URL http://www.sciencedirect.com/science/article/pii/S0306437910000025.

Benavides, D., Segura, S., Trinidad, P., Cortés, A.R., 2007. FAMA: Tooling a framework for the automated analysis of feature models. In: First International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS, pp. 129–134.

Berger, T., Collet, P., 2019. Usage scenarios for a common feature modeling language. In: 23rd International Systems and Software Product Line Conference, Vol. B. SPLC, ACM, pp. 86:1–86:8. http://dx.doi.org/10.1145/3307630.3342403.

Berger, T., Pfeiffer, R., Tartler, R., Dienst, S., Czarnecki, K., Wasowski, A., She, S., 2014. Variability mechanisms in software ecosystems. Inf. Software Technol. 56 (11), 1520–1535. http://dx.doi.org/10.1016/j.infsof.2014.05.005.

Berger, T., She, S., Lotufo, R., Wasowski, A., Czarnecki, K., 2013. A study of variability models and languages in the systems software domain. IEEE Trans. Software Eng. 39 (12), 1611–1640. http://dx.doi.org/10.1109/TSE.2013.34.

Butting, A., Eikermann, R., Kautz, O., Rumpe, B., Wortmann, A., 2018a. 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, ACM, pp. 75–82. http://dx.doi.org/10.1145/3168365.3168368.

Butting, A., Eikermann, R., Kautz, O., Rumpe, B., Wortmann, A., 2018b. Modeling language variability with reusable language components. In: 22nd International Systems and Software Product Line Conference, Vol. 1. SPLC, ACM, pp. 65–75. http://dx.doi.org/10.1145/3233027.3233037.

Cañete, A., Amor, M., Fuentes, L., 2020. Supporting the evolution of applications deployed on edge-based infrastructures using multi-layer feature models. In: SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume B. pp. 79–87. http://dx.doi.org/10.1145/3382026.3425772.

Classen, A., Boucher, Q., Heymans, P., 2011. A text-based approach to feature modelling: Syntax and semantics of TVL. Sci. Comput. Program. 76 (12), 1130–1143. http://dx.doi.org/10.1016/j.scico.2010.10.005.

Cordy, M., Schobbens, P.-Y., Heymans, P., Legay, A., 2013. Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13, IEEE Press, Piscataway, NJ, USA, pp. 472–481, URL http://dl.acm.org/citation.cfm?id=2486788.2486851.

Czarnecki, K., Eisenecker, U.W., 2000. Generative Programming - Methods, Tools and Applications. Addison-Wesley, URL http://www.addison-wesley.de/main/main.asp?page=englisch/bookdetails&productid=99258.

Czarnecki, K., Helsen, S., Eisenecker, U.W., 2005. Formalizing cardinality-based feature models and their specialization. Software Process: Improv. Pract. 10 (1), 7–29. http://dx.doi.org/10.1002/spip.213.

Eichelberger, H., Kröher, C., Schmid, K., 2013. An analysis of variability modeling concepts: Expressiveness vs. analyzability. In: 13th International Conference on Software Reuse. ICSR, In: Lecture Notes in Computer Science, vol. 7925, Springer, pp. 32–48. http://dx.doi.org/10.1007/978-3-642-38977-1_3.

Fadhlillah, H.S., Feichtinger, K., Sonnleithner, L., Rabiser, R., Zoitl, A., 2021. Towards heterogeneous multi-dimensional variability modeling in cyber-physical production systems. In: SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kindom, September 6-11, 2021, Volume B. ACM, pp. 123–129. http://dx.doi.org/10.1145/3461002.3473941.

Feichtinger, K., 2021. A flexible approach for transforming variability models. In: SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kindom, September 6-11, 2021, Volume B. ACM, pp. 18–23. http://dx.doi.org/10.1145/3461002.3473069.

Feichtinger, K., Rabiser, R., 2020. Variability model transformations: Towards unifying variability modeling. In: 46th Euromicro Conference on Software Engineering and Advanced Applications. SEAA 2020, Portoroz, Slovenia, August 26-28, 2020, IEEE, pp. 179–182. http://dx.doi.org/10.1109/SEAA51224.2020.00037.

Feichtinger, K., Rabiser, R., 2021. How flexible must a transformation approach for variability models and custom variability representations be? In: SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kindom, September 6-11, 2021, Volume B. ACM, pp. 69–72. http://dx.doi.org/10.1145/3461002.3473945.

Feichtinger, K., Stöbich, J., Romano, D., Rabiser, R., 2021. TRAVART: An approach for transforming variability models. In: VaMoS'21: 15th International Working Conference on Variability Modelling of Software-Intensive Systems, Virtual Event / Krems. Austria, February 9-11, 2021, ACM, pp. 8:1–8:10. http://dx.doi.org/10.1145/3442391.3442400.

Galindo, J.A., Benavides, D., Trinidad, P., Gutiérrez-Fernández, A.M., Ruiz-Cortés, A., 2019. Automated analysis of feature models: Quo vadis? Computing 101 (5), 387–433. http://dx.doi.org/10.1007/s00607-018-0646-1.

Galster, M., Weyns, D., Tofan, D., Michalik, B., Avgeriou, P., 2014. Variability in software systems - A systematic literature review. IEEE Trans. Software Eng. 40 (3), 282–306. http://dx.doi.org/10.1109/TSE.2013.56.

Gheyi, R., Massoni, T., Borba, P., 2011. Automatically checking feature model refactorings. J. UCS 17 (5), 684–711. http://dx.doi.org/10.3217/jucs-017-05-0684.

Harel, D., Rumpe, B., 2004. Meaningful modeling: What's the semantics of "semantics"? IEEE Comput. 37 (10), 64–72. http://dx.doi.org/10.1109/MC.2004.172.

Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A., 2008. Adding standardized variability to domain specific languages. In: 12th International Software Product Line Conference. SPLC 2008), IEEE Computer Society, pp. 139–148. http://dx.doi.org/10.1109/SPLC.2008.25.

Heradio, R., Perez-Morago, H., Fernández-Amorós, D., Bean, R., Cabrerizo, F.J., Cerrada, C., Herrera-Viedma, E., 2016. Binary decision diagram algorithms to perform hard analysis operations on variability models. In: New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Fifteenth SoMeT_16. Larnaca, Cyprus, 12-14 September 2016, In: Frontiers in Artificial Intelligence and Applications, vol. 286, IOS Press, pp. 139–154. http://dx.doi.org/10.3233/978-1-61499-674-3-139.

Heymans, P., Schobbens, P.-Y., Trigaux, J.-C., Matulevicius, R., Classen, A., Bontemps, Y., 2007. Towards the comparative evaluation of feature diagram languages. In: Software and Services Variability Management Workshop Concepts, Models and Tools. SVM 2007.

Horcas, J.M., Galindo, J.A., Heradio, R., Fernandez-Amoros, D., Benavides, D., 2023. A monte carlo tree search conceptual framework for feature model analyses. 195, p. 111551. http://dx.doi.org/10.1016/j.jss.2022.111551,

Horcas, J.M., Pinto, M., Fuentes, L., 2020. Extensible and modular abstract syntax for feature modeling based on language constructs. In: 24th International Systems and Software Product Line Conference. SPLC, ACM, http://dx.doi.org/10.1145/3382025.3414959.

Horcas, J.M., Pinto, M., Fuentes, L., 2022. Empirical analysis of the tool support for software product lines. Software Syst. Model. http://dx.doi.org/10.1007/s10270-022-01011-2.

Juodisius, P., Sarkar, A., Mukkamala, R.R., Antkiewicz, M., Czarnecki, K., Wasowski, A., 2019. Clafer: Lightweight modeling of structure, behaviour, and variability. Program. J. 3 (1), 2. http://dx.doi.org/10.22152/programming-journal.org/2019/3/2.

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M., 1998. FORM: A feature-oriented reuse method with domain-specific reference architectures. Ann. Software Eng. 5, 143–168. http://dx.doi.org/10.1023/A:1018980625587.

Knüppel, A., Thüm, T., Mennicke, S., Meinicke, J., Schaefer, I., 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, ACM, pp. 291–302. http://dx.doi.org/10.1145/3106237.3106252.

Krüger, J., Nielebock, S., Krieter, S., Diedrich, C., Leich, T., Saake, G., Zug, S., Ortmeier, F., 2017. Beyond software product lines: Variability modeling in cyber-physical systems. In: Proceedings of the 21st International Systems and Software Product Line Conference - Volume. A. SPLC '17, ACM, New York, NY, USA, pp. 237–241. http://dx.doi.org/10.1145/3106195.3106217, URL http://doi.acm.org/10.1145/3106195.3106217.

Liang, J.H., Ganesh, V., Czarnecki, K., Raman, V., 2015. SAT-based analysis of large real-world feature models is easy. In: Proceedings of the 19th International Conference on Software Product Line. SPLC '15, Association for Computing Machinery, New York, NY, USA, pp. 91–100. http://dx.doi.org/10.1145/2791060.2791070.

Liu, F., Tang, G., Li, Y., Cai, Z., Zhang, X., Zhou, T., 2019. A survey on edge computing systems and tools. Proc. IEEE 107 (8), 1537–1562.

Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., Saake, G., 2017. Mastering Software Variability with FeatureIDE. Springer, http://dx.doi.org/10.1007/978-3-319-61443-4.

Meixner, K., Rabiser, R., Biffl, S., 2019. Towards modeling variability of products, processes and resources in cyber-physical production systems engineering. In: 23rd International Systems and Software Product Line Conference, Vol. B. SPLC, ACM, pp. 68:1–68:8. http://dx.doi.org/10.1145/3307630.3342411.

Merenda, M., Porcaro, C., Iero, D., 2020. Edge machine learning for AI-enabled IoT devices: A review. Sensors 20 (9), 2533. http://dx.doi.org/10.3390/s20092533.

Munoz, D., Gurov, D., Pinto, M., Fuentes, L., 2021. Category theory framework for variability models with non-functional requirements. In: Advanced Information Systems Engineering - 33rd International Conference, CAiSE 2021, Melbourne, VIC, Australia, June 28 - July 2, 2021, Proceedings. In: Lecture Notes in Computer Science, vol. 12751, Springer, pp. 397–413. http://dx.doi.org/10.1007/978-3-030-79382-1_24.

Munoz, D., Oh, J., Pinto, M., Fuentes, L., Batory, D.S., 2019. Uniform random sampling product configurations of feature models that have numerical features. In: 23rd International Systems and Software Product Line Conference, Vol. A. SPLC, ACM, pp. 39:1–39:13. http://dx.doi.org/10.1145/3336294.3336297.

Object Management Group (OMG), 2016. Meta object facility (MOF), v2.5.1. https://www.omg.org/spec/MOF/Current.

Pohl, K., Böckle, G., van der Linden, F., 2005. Software Product Line Engineering - Foundations, Principles, and Techniques. Springer, http://dx.doi.org/10.1007/3-540-28901-1.

Raatikainen, M., Tiihonen, J., Männistö, T., 2019. Software product lines and variability modeling: A tertiary study. J. Syst. Softw. 149, 485–510. http://dx.doi.org/10.1016/j.jss.2018.12.027.

Romero, D., Galindo, J.A., Horcas, J.M., Benavides, D., 2021. A first prototype of a new repository for feature model exchange and knowledge sharing. In: SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kindom, September 6-11, 2021, Volume B. ACM, pp. 80–85. http://dx.doi.org/10.1145/3461002.3473949.

Rosenmüller, M., Siegmund, N., Thüm, T., Saake, G., 2011. Multi-dimensional variability modeling. In: 5th International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS, In: ACM International Conference Proceedings Series, ACM, pp. 11–20. http://dx.doi.org/10.1145/1944892.1944894.

Schmid, K., Kröher, C., El-Sharkawy, S., 2018. Variability modeling with the integrated variability modeling language (IVML) and EASy-producer. In: 22nd International Systems and Software Product Line Conference, Vol. 1. SPLC 2018, ACM, p. 306. http://dx.doi.org/10.1145/3233027.3233057.

Schmid, K., Rabiser, R., Grünbacher, P., 2011. A comparison of decision modeling approaches in product lines. In: 5th International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS, In: ACM International Conference Proceedings Series, ACM, pp. 119–126. http://dx.doi.org/10.1145/1944892.1944907.

Schmitt, A., Bettinger, C., Rock, G., 2018. Glencoe – A tool for specification, visualization and formal analysis of product lines. In: 25th International Conference on Transdisciplinary Engineering. In: Advances in Transdisciplinary Engineering, pp. 665–673. http://dx.doi.org/10.3233/978-1-61499-898-3-665.

Schobbens, P., Heymans, P., Trigaux, J., Bontemps, Y., 2007. Generic semantics of feature diagrams. Comput. Networks 51 (2), 456–479. http://dx.doi.org/10.1016/j.comnet.2006.08.008.

Seidl, C., Winkelmann, T., Schaefer, I., 2016. A software product line of feature modeling notations and cross-tree constraint languages. In: Modellierung. In: LNI, vol. P-254, GI, pp. 157–172, URL https://dl.gi.de/20.500.12116/821.

Sepúlveda, S., Cares, C., Cachero, C., 2012. Towards a unified feature metamodel: A systematic comparison of feature languages. In: 7th Iberian Conference on Information Systems and Technologies. CISTI, pp. 1–7.

Sepúlveda, S., Cravero, A., Cachero, C., 2016. Requirements modeling languages for software product lines: A systematic literature review. Inf. Softw. Technol. 69, 16–36. http://dx.doi.org/10.1016/j.infsof.2015.08.007.

Steinberg, D., Budinsky, F., Merks, E., Paternostro, M., 2008. EMF: Eclipse Modeling Framework. Pearson Education.

Sundermann, C., Feichtinger, K., Engelhardt, D., Rabiser, R., Thüm, T., 2021. Yet another textual variability language?: A community effort towards a unified language. In: SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A. ACM, pp. 136–147. http://dx.doi.org/10.1145/3461001.3471145.

Sundermann, C., Thüm, T., Schaefer, I., 2020. Evaluating #SAT solvers on industrial feature models. In: 14th International Working Conference on Variability Modelling of Software-Intensive Systems. VaMoS, ACM, pp. 3:1–3:9. http://dx.doi.org/10.1145/3377024.3377025.

Tanhaei, M., Habibi, J., Mirian-Hosseinabadi, S., 2016. Automating feature model refactoring: A model transformation approach. Inf. Softw. Technol. 80, 138–157. http://dx.doi.org/10.1016/j.infsof.2016.08.011.

ter Beek, M.H., Schmid, K., Eichelberger, H., 2019. Textual variability modeling languages: An overview and considerations. In: 23rd International Systems and Software Product Line Conference (SPLC), Volume B. ACM, pp. 82:1–82:7. http://dx.doi.org/10.1145/3307630.3342398.

Thüm, T., Batory, D.S., Kästner, C., 2009. Reasoning about edits to feature models. In: 31st International Conference on Software Engineering. ICSE, IEEE, pp. 254–264. http://dx.doi.org/10.1109/ICSE.2009.5070526.

Thüm, T., Seidl, C., Schaefer, I., 2019. On language levels for feature modeling notations. In: 23rd International Systems and Software Product Line Conference, Vol. B. SPLC, ACM, pp. 83:1–83:4. http://dx.doi.org/10.1145/3307630.3342404.

Urli, S., Blay-Fornarino, M., Collet, P., Mosser, S., 2012. Using composite feature models to support agile software product line evolution. In: Proceedings of the 6th International Workshop on Models and Evolution. ME@MoDELS 2012, Innsbruck, Austria, October 1-5, 2012, pp. 21–26. http://dx.doi.org/10.1145/2523599.2523604.

Villota, Á., Mazo, R., Salinesi, C., 2019. The high-level variability language: An ontological approach. In: 23rd International Systems and Software Product Line Conference, Vol. B. SPLC, ACM, pp. 84:1–84:8. http://dx.doi.org/10.1145/3307630.3342401.

Wortmann, A., Barais, O., Combemale, B., Wimmer, M., 2020. Modeling languages in industry 4.0: An extended systematic mapping study. Softw. Syst. Model. 19 (1), 67–94. http://dx.doi.org/10.1007/s10270-019-00757-6.

Zhiyi, M., Xiao, H., 2014. Building modeling tools based on metamodeling and product line technologies. Chin. J. Electron. 23 (2).

José Miguel Horcas is a postdoc researcher at the University of Málaga, Spain, where he received his M.Sc. degree in 2012 and his Ph.D. in Computer Sciences in 2018. He is a member of the CAOSD research group of the University of Málaga. He carried out two postdoc stays at King's College London in 2019 for three months, and at the University of Seville for 18 months. His main research areas are related to software product lines, including variability and configurability, and quality attributes. More information available at https://sites.google.com/view/josemiguelhorcas.

Mónica Pinto received the M.Sc. degree in computer science and the Ph.D. degree from the Universidad de Málaga, Spain, in 1998 and 2004, respectively. She is an Associate Professor since 2009 with the Department of Lenguajes y Ciencias de la Computación, Universidad de Málaga. She is a research member of the CAOSD research group, one of the constituent group of the Instituto de Tecnología e Ingeniería del Software "José María Troya Linero of the Universidad de Málaga. She is currently part of the institute management team. She actively participates in Spanish and European research projects. Her main research areas are energy-aware software development, quality-driven variability modeling and analysis, model-driven software engineering, and Internet Of Things and Edge computing systems development.

Lidia Fuentes received her M.Sc. degree in Computer Science from the University of Málaga (Spain) in 1992 and her Ph.D. in Computer Science in 1998 from the same University. She is a Full Professor at the Department of Computer Science of the University of Málaga since 2011 (previously, Lecturer and Associate Professor from 1993). Currently, she is the head of the CAOSD research group. Her main research areas are Aspect-Oriented Software Development, Model-Driven Development, Software Product Lines, AgentOriented Software Engineering, Self-adaptive middleware platforms, Architecture Description Languages and Domain Specific Languages.