Accepted manuscript at Software and Systems Modeling manuscript No.
(will be inserted by the editor)

FLAME: a Formal Framework for the Automated Analysis of Software Product Lines Validated by Automated Specification Testing

Amador Durán · David Benavides · Sergio Segura · Pablo Trinidad · Antonio Ruiz–Cortés

The final publication is available at Springer via http://dx.doi.org/10.1007/s10270-015-0503-z

Abstract In a literature review on the last 20 years of automated analysis of feature models, the formalization of analysis operations was identified as the most relevant challenge in the field. This formalization could provide very valuable assets for tool developers such as a precise definition of the analysis operations and, what is more, a *reference implementation*, i.e. a trustworthy, not necessarily efficient implementation to compare different tools outputs. In this article, we present the FLAME framework as the result of facing this challenge. FLAME is a formal framework that can be used to formally specify not only feature models, but other *variability modeling languages* (VMLs) as well. This reusability is achieved by its two–layered architecture. The *abstract foundation layer* is the bottom layer in which all VML–independent analysis operations and concepts are specified. On top of the foundation layer, a family of *characteristic model layers*—one for each VML to be formally specified—can be developed by redefining some abstract types and relations. The verification and validation of FLAME has followed a process in which formal verification has been performed traditionally by manual theorem proving, but validation has been performed by integrating our experience on *metamorphic testing* of variability analysis tools, something that has shown to be much more effective than manually–designed test cases. To follow this automated, test–based validation approach, the specification of FLAME, written in Z, was translated into Prolog and 20,000 random tests were automatically generated and executed. Tests results helped to discover some inconsistencies not only in the formal specification, but also in the previous informal definitions of the analysis operations and in current analysis tools. After this process, the Prolog implementation of FLAME is being used as a reference implementation for some tool developers, some analysis operations have been formally specified for the first time with more generic semantics, and more VMLs are being formally specified using FLAME.

Keywords Formal Specification · Specification Testing · Software Product Lines · Feature Models

This work was partially supported by the European Commission (FEDER), and the Spanish and Andalusian R&D&I program grants COPAS (P12–TIC–1867), TAPAS (TIN2012–32273), THEOS (TIC–5906), and SaaS Firewall (IPT–2013–0890–3).

A. Durán (✉) · D. Benavides · S. Segura · P. Trinidad · A. Ruiz–Cortés
Department of Computer Languages and Systems
Universidad de Sevilla, E.T.S.I. Informática,
Av. Reina Mercedes s/n, 41012, Seville, Spain
E-mail: amador@us.es

D. Benavides
E-mail: benavides@us.es

S. Segura
E-mail: sergiosegura@us.es

P. Trinidad
E-mail: ptrinidad@us.es

A. Ruiz–Cortés
E-mail: aruiz@us.es

1 Introduction

The *variability* of a software–intensive system can be defined as the capability of being tailored or adapted to specific needs in a specific domain. *Software product lines* (SPLs) are commonly used as a way of managing the variability of a family of similar software systems in a concrete domain. In the SPL context, software variability is usually documented using *variability modeling languages* (VMLs), which describe all the possible configurations of a software system in terms of (1) composable units or *variants*, and (2) *constraints* indicating how those variants can be properly combined. At the *problem* level, variability is modeled in terms of *features* or requirements, usually using *feature models*

(FMs) [39]. On the other hand, at the *solution* level variability is modeled using domain–specific languages such as *Kconfig* in Linux [11], *p2* in Eclipse [41] or *WS–Agreement* in web services [45].

In outline, SPL engineering covers specific processes, methods, models, techniques and tools for supporting SPL adoption [19,51]. As an essential support for SPL engineers during domain analysis, the automated analysis of FMs is defined in [9] as the computer–aided extraction of information from FMs by means of *analysis operations*, such as determining the number of products represented by a model, detecting model anomalies, etc. Manual computation of such analysis operations is error–prone, tedious, and even infeasible with large–scale FMs.

Some of the authors performed a systematic literature review on the last 20 years of automated analysis of FMs in which 30 different analysis operations were cataloged [9]. This review also identified several important challenges that were not covered by existing research. One of them was the lack of formal or rigorous descriptions of analysis operations, which has sometimes led researchers and tool developers to misunderstandings. Notice that a precise definition of the analysis operations and, what is more, a *reference implementation*, i.e. a trustworthy, not necessarily efficient implementation of the analysis operations to compare different tools outputs, are very valuable assets for tool developers.

This article, which is mainly focused on *problem–level* variability, presents FLAME (*FaMa formaL frAMEwork*), a formal framework for specifying the semantics of analysis operations not only on FMs, but also on other VMLs. FLAME is architectured in two layers. The bottom layer is the *abstract foundation layer* (AFL), which includes the definitions of necessary abstract concepts that can or must be redefined in the second layer, and also 20 VML–independent analysis operations. On top on the AFL, a family of *characteristic model layers* (CMLs)—one for each VML to be formally specified—can be developed by redefining the aforementioned abstract concepts (see Figure 1). In this article, a CML specifying the semantics of an eclectic FM dialect known as *basic feature model* (BFM) [9] is presented, although other VMLs like OVM [51] or CUDF [69], which is used for package–based Linux distributions, can also be specified (see appendices A and B for an overview).

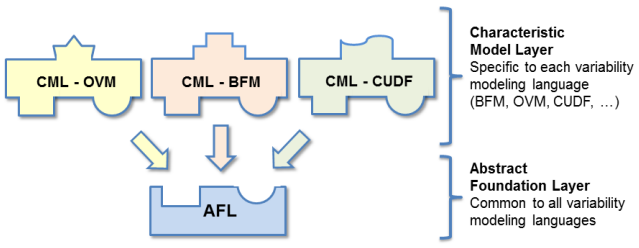


Fig. 1 FLAME architecture: relation between layers

During the development of FLAME, the verification and validation processes followed an approach in which formal verification were performed traditionally by manual theorem specification and proving, but validation were performed by integrating our experience on *metamorphic testing* of variability analysis tools [64,62], which has shown to be much more effective than manually–designed test cases, as described in [61]. To follow this automated, test–based validation approach, the specification of FLAME –developed in the standard, highly–expressive, well–known Z formal specification language [65,37]—was translated into Prolog [20] for testing purposes and 20,000 metamorphic random tests were automatically generated and executed. Traditionally, Prolog [20] has been the choice for the testing of Z specifications, also know as *specification animation* [35,73]. In the case of FLAME, animating the Z specification in Prolog was very useful for detecting problems and inconsistencies by the manual execution of a small suite of tests, which were very helpful during the discussions among the authors about the semantics of some operations and provided immediate feedback. On the other hand, the animation was also systematically tested against 20,000 random test cases automatically generated using *metamorphic testing* techniques inspired by the previous work of some of the authors [64,62]. Metamorphic testing [15], exploits the relations between the inputs and outputs of a program to generate new *follow–up* test cases from existing test data (see Section 2.3 for an overview and Section 5.1 for details).

After this exhaustive testing, the Prolog animation can be considered as a high–level reference implementation for tool developers that is not designed for efficiency but to be easy to understand and to clarify the semantics of many analysis operations that had not been formally specified before. The integrated approach —formal and test–based at the same time— has helped to discover some inconsistencies not only in the formal specification but also in the previous informal definitions of the analysis operations in [9] and in current analysis tools such as the FaMa framework [10,36].

The reminder of the article is structured as follows. Section 2 provides the necessary background on FMs, their automated analysis, and metamorphic testing for those readers not familiar with the topics; Section 3 describes the AFL layer of the FLAME framework, including the theorems used for the formal verification of the specification; Section 4 describes a concrete application of the CML of FLAME to a specific VML, namely BFM; Section 5 describes the test–based validation of the specification and their results; Section 6 comments the related work and finally, Section 7 presents the conclusions and the future work.

Four appendices with supplemental material are available for the reader. Appendices A and B contain an overview of the specification in FLAME of the OVM [51] and CUDF [69] VMLs respectively. Appendix C contains the proofs of

the theorems included in Section 3. Finally, Appendix D includes the guidelines applied for the manual translation of the Z specification into executable Prolog code, and an example of use of the reference implementation, which can be downloaded from `http://www.isa.us.es/flame`.

2 Background

2.1 Feature models

As mentioned in the previous section, FMs are widely used to describe the set of products in an SPL in terms of their *features*. In these models, features are hierarchically linked in a tree–like structure and are optionally connected by cross–tree constraints. An example on how FMs are usually depicted using FODA–like notations [39] is shown in Figure 2, where the FM describes an SPL for mobile phones borrowed from [9].

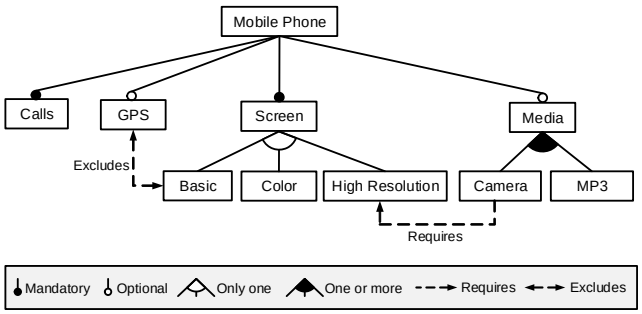


Fig. 2 A sample feature model of an SPL for mobile phones using a FODA–like notation

Although there are many proposals on the type of relationships and their graphical representation in FMs (see [59] for a detailed survey), the most usual relationships inspired in the seminal work by Kang *et al.* [39] are the following:

- *Mandatory*: a *child feature* has a *mandatory* relationship with its *parent feature* when it is required to appear in a given product whenever its parent feature appears in that product. In Figure 2, Calls has a mandatory relationship with Mobile Phone, i.e. any product in the mobile phone SPL must have a feature to manage calls.
- *Optional*: a child feature has an *optional* relationship with its parent feature when it can appear or not in a given product whenever its parent feature appears in that product. In the example in Figure 2, GPS has an optional relationship with Mobile Phone, i.e. the GPS feature can be optionally chosen in the configuration of a product in the mobile phone SPL.
- *Or–relationship* (also known as *OneOrMore*): a set of child features have an *or–relationship* with their parent feature when one or more child features can be selected in a given product when the parent feature appears in that product. In Figure 2, Camera and MP3 has an or–relationship with Media, which means that whenever Media is selected, Camera, MP3, or both must be selected.
- *Alternative* (also known as *OnlyOne*): a set of child features have an *alternative* relationship with their parent feature when only one of them must be selected in a given product when their parent feature appears in that product. Features Basic, Color and High Resolution have an alternative relationship in Figure 2, where one and only one of them must be selected whenever Screen is present in a product.
- *Requires*, *Excludes*: a cross–tree relationship like A requires B means that in any product where feature A appears, feature B must also appear. On the other hand, a relationship like A excludes B means that both features cannot appear in the same product at the same time. In the example in Figure 2, Camera requires High Resolution, whereas GPS excludes a Basic screen.

2.2 Automated analysis of feature models

As commented in the introductory section, the automated analysis of FMs deals with the computer–aided extraction of information from FMs. From the information obtained, SPL engineers can decide marketing strategies and make technical decisions. Many different analysis operations on FMs have been reported in the literature [9] in the last years. Some of the most referenced are presented below to provide an overview for those readers not familiar with the topic.

- *Finding out if a product is valid.* This operation checks whether an input product (i.e. set of features) belongs to the set of products represented by a given FM or not. It may be helpful for SPL engineers and managers to determine whether a given product is available in an SPL.
- *Obtaining all products.* This operation takes an FM as input and returns all the products represented by the model. It may be helpful to identify new valid requirement combinations not considered in the initial scope of the SPL.
- *Calculating the number of products.* This operation returns the number of products represented by an FM, which provides information about the flexibility and complexity of an SPL. The number of products is used in derived operations such as variability, commonality and homogeneity metrics of an FM (see Section 3.5).
- *Determining if a FM is void.* This operation takes an FM and determines whether the FM is void or not, i.e. whether it represents no products. The automation of this operation is especially helpful when debugging large–scale FMs in which the manual detection of errors is an error–prone, time–consuming task.

– *Detecting dead features.* This operation takes an FM as input and returns the set of dead features included in the model. A feature is *dead* if it cannot appear in any of the products derived from the model. Dead features are usually caused by a wrong usage of cross–tree constraints and are clearly undesired since they are the result of a wrong domain modeling.

These operations can be performed automatically using different approaches. Most translate FMs into specific logic paradigms such as propositional logic, constraint programming or description logic. Others propose *ad–hoc* algorithms and solutions to perform these analyses [9]. Finally, these analysis capabilities can also be found in several commercial and open source tools such as the *AHEAD Tool Suite* [3], the *Big Lever Software Gears* [12], the *FaMa Framework* [36], the *Feature Model Plug–in* [28], *pure::variants* [52], and *SPLOT* [43]. The automated analysis of FMs is being used in different scenarios such as cloud computing configurations [31], reverse engineering of feature models [42], image test case selections [29] or testing of mobile phone configurations [30].

2.3 Metamorphic testing

In software testing, an *oracle* is a procedure by which testers can decide whether the output of a program is correct or not [74]. In some situations, the oracle is not available or it is too difficult to apply. For example, consider testing the results of complicated numerical computations such as the Fourier transform, or processing non–trivial outputs like the code generated by a compiler. Furthermore, even when the oracle is available, the manual prediction and comparison of the results are in most cases time–consuming and error–prone. Situations like these are referred to as the *oracle problem* in the testing literature [80].

Metamorphic testing [15] was proposed as a way to address the oracle problem by generating new tests from previously successful test cases. The expected output of the new test cases can be checked by using so–called *metamorphic relations*, i.e. known relations among two or more input data and their expected outputs. As a result, the oracle problem is alleviated and the test data generation process can be highly automated. For instance, consider a program that compute the sine function ($\sin x$). Suppose the program produces the output 0.207 when run with input $x = 12$. A mathematical property of the sine function states that $\sin(x) = \sin(x + 360)$. Using this property as a metamorphic relation, a new test case with $x = 12 + 360 = 372$ could be designed. Assuming that the output of the program for this input is 0.375, it could be easily concluded that the program is faulty by comparing both outputs.

Metamorphic testing has been successfully applied to a number of testing domains including numerical programs [16], graph theory [17], service–oriented applications [14], or variability analysis tools [62]. In this article, it is applied to the automated analysis of FMs (see Section 5.1).

3 Abstract foundation layer of the FLAME framework

The *abstract foundation layer* (AFL) is the basement on which the FLAME framework rests. In the AFL, some abstract concepts and most of the analysis operations described in [9] are formally defined. Notice that although those operations were originally defined over FMs, it has been possible to remove their dependencies from that VML and transform most of them into generic SPL analysis operations.

In the rest of this section, all the concepts and operations related to SPL analysis from an abstract point of view are described informally in natural language but also formally in Z. More specifically, in section 3.1, the basic SPL concepts, their integration in the framework architecture, and their representation in Z, are discussed. In section 3.2, the basic SPL analysis operations (product validity, the set of valid products, void SPL checking, etc.) are specified. In section 3.3, SPL relations such as equivalence and specialization are defined. In section 3.4, feature–related operations (core features, dead features, etc.) are formalized. Finally, in section 3.5, some numerical indicators such as commonality, variability and homogeneity are specified.

The adopted name convention in the formal specification is that operations related to *features* use the Greek letter *phi* (Φ), those related to *products* use the Greek letter *pi* (Π), and operations yielding numbers use calligraphic letters such as $\mathcal{N}$. When appropriate, some theorems used during formal verification are included after operation specifications. Theorem proofs can be found in Appendix C.

3.1 SPL basic concepts

From an abstract point of view, an SPL can be considered as composed of two elements: (1) a nonempty set of *features* that can be combined to form *products*; and (2) a *characteristic model* which determines which of those combinations are *valid products* of the SPL. More formally, a *product*, considered as a finite nonempty set of features, is a valid product of an SPL if: (1) its set of features is a subset of the SPL feature set, i.e. it contains only *known features*; and (2) if it is an *instance* of the characteristic model of the SPL.[1]

[1] As defined by D. Batory [5], this is very similar to the concept of formal languages. In SPLs, the alphabet is the set of features, the grammar is the characteristic model, and the language is the set of all products that are instances of the characteristic model.

Notice that in order to keep the AFL abstract and therefore reusable, nothing is said about the nature of either *features* or *models*, which are open for redefinition in the CML of FLAME when needed. For example, *features* could be redefined to include attributes, whereas characteristic *models* and their associated *is–instance–of* relations must be redefined in the CML in order to specify the semantics of a concrete VML like FMs, OVM [51] or CUDF [69]. The function computing the *valid products* of an SPL can also be redefined in the CML, something interesting when dealing with a big number of features and an efficient notation–dependent algorithm is known, since the default specification of the valid products function has an exponential algorithmic complexity (see Section 3.2.2 for details).

A summary of the redefinable abstract concepts defined in the AFL, indicating whether they must be redefined in the CML or not, is shown in Table 1. Also, a UML class diagram representing these concepts from an object–oriented point of view is shown in Figure 3, i.e. the AFL can be seen as an abstract package in which some abstract methods must or can be overridden in subclasses.

Table 1 Redefinable abstract concepts in the Abstract Foundation Layer of FLAME

Name	Description	Redef. in CML
SPL	Type and invariant for SPLs	optional
$Feature$	Type for features	optional
$Model$	Type for VML characteristic models	mandatory
$\_ \prec\!\!\!\cdot \_$	*is–instance–of* relation	mandatory
Φ	*features–in–a–model* function	mandatory
Π	*valid–products* function	optional

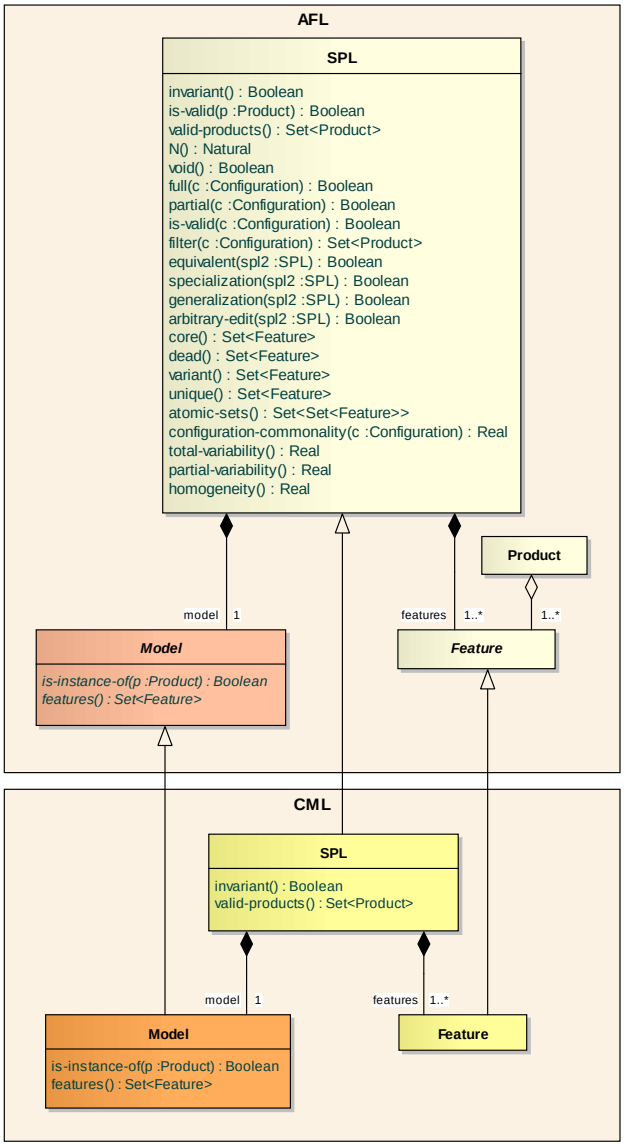


Fig. 3 UML class diagram of the FLAME architecture

3.1.1 SPL basic concepts in Z

Features, models, and products. To express the previously mentioned concepts in Z, the two abstract types $Feature$ and $Model$ —i.e. *given sets* in Z terminology, see [65] for details— are defined. Then, the $Product$ type is defined as a finite nonempty set of features[2].

$$[Feature] \qquad \text{[Abstract type for features]}$$
$$[Model] \qquad \text{[Abstract type for characteristic models]}$$
$$Product == \mathbb{F}_1\ Feature \qquad \text{[Product type]}$$

[2] In Z, $\mathbb{P}\ S$ denotes the *powerset* of the set S, containing all possible subsets of S, even the infinite ones. On the other hand, $\mathbb{F}\ S$ denotes the *finite powerset* of S, containing finite subsets only. If the empty set is excluded, the notation becomes $\mathbb{P}_1$ and $\mathbb{F}_1$. Notice that if S is finite, $\mathbb{P}\ S$ and $\mathbb{F}\ S$ are the same.

Configurations. Another useful concept for SPL automated analysis, especially during product design, is the so–called *configuration* [9], in which the customer can select the features she wants in the final product, remove undesired features, and leave other features as undecided. Formally, a configuration defined over a set of features of a given SPL is a pair of disjoint subsets indicating the features to be *selected* and to be *removed*. This can be defined in Z as follows:

$$Configuration : \mathbb{F}\ Feature \times \mathbb{F}\ Feature$$
$$selected, removed : Configuration \to \mathbb{F}\ Feature$$
$$\forall\ c : Configuration \bullet$$
$$selected\ c = first\ c \land$$
$$removed\ c = second\ c \land$$
$$selected\ c \cap removed\ c = \varnothing$$

where first and second are standard Z functions to respectively access the first and second elements of any pair of objects in a Cartesian product. Depending on whether some features are left undecided or not, a configuration is said to be *full* or *partial* with respect to an SPL:

$$\mathit{full}, \mathit{partial} : \mathit{Configuration} \times \mathit{SPL}$$

$$\begin{array}{l} \forall\, c : \mathit{Configuration};\ \mathit{spl} : \mathit{SPL} \bullet \\ \quad \mathit{full}(\ c, \mathit{spl}\ ) \Leftrightarrow \\ \qquad (\mathit{selected}\ c \cup \mathit{removed}\ c) = \mathit{spl}.\mathit{features} \\ \quad \wedge \\ \quad \mathit{partial}(\ c, \mathit{spl}\ ) \Leftrightarrow \\ \qquad (\mathit{selected}\ c \cup \mathit{removed}\ c) \subset \mathit{spl}.\mathit{features} \end{array}$$

For example, a possible partial configuration for the SPL represented by the FM in Figure 2 could be ({Mobile Phone, Calls, Screen, Color, Media, MP3}, {GPS}), indicating that the selected features are a mobile phone with calls, a color screen, and able to play MP3 media files, whereas GPS support is not desired. All other features are left undecided. See Sections 3.2.5 and 3.2.6 for analysis operations related to configurations.

Instance of a model. Once the $\mathit{Product}$ and Model types are defined, the abstract *is–instance–of* relation between products and models (denoted as $\prec\!\!\!\prec$, a symbol borrowed from the Object–Z notation [40]) can also be defined as follows:

$$\_ \prec\!\!\!\prec \_ : \mathit{Product} \leftrightarrow \mathit{Model}$$

$$\begin{array}{l} \forall\, p : \mathit{Product};\ m : \mathit{Model} \bullet \\ \quad p \prec\!\!\!\prec m \Leftrightarrow [\text{ p is an instance of m }] \\ \quad [\text{concrete definition must be provided in the CML}] \end{array}$$

where the concrete definition of the relation and the Model type must be provided in the CML corresponding to the VML being specified (see Table 1). For example, the product {Mobile Phone, Calls, Screen, Basic} is an instance of the FM in Figure 2, whereas {Mobile Phone, Calls, Screen, Basic, GPS} is not because GPS and Basic features exclude each other.

Features in a model. An abstract function returning the set of features used in a given characteristic model needs also to be defined in order to specify the constraint that all the features in an SPL must be involved in its characteristic model and vice versa, i.e. that an SPL cannot contain *unbound* features and that a characteristic model must use *all and only* the features in its SPL. This abstract function (denoted as Φ) is defined as follows:

$$\Phi : \mathit{Model} \rightarrow \mathbb{F}\ \mathit{Feature}$$

[concrete definition must be provided in the CML]

where the concrete definition, as for the $\prec\!\!\!\prec$ relation, must also be provided in the CML corresponding to the VML being specified.

SPL as a type. Finally, using the previous definitions, an abstract SPL can be formally defined as the following *schema type* in Z:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{SPL}} \\ \mathit{model} : \mathit{Model} & [\text{SPL characteristic model}] \\ \mathit{features} : \mathbb{F}_1\ \mathit{Feature} & [\text{SPL feature set}] \\ \hline \Phi\ \mathit{model} = \mathit{features} & \\ [\text{ other invariants can be added in the CML }] & \end{array}$$

which is considered as abstract in order to be augmented with additional invariants on features or models when used in the CML for the specification of concrete VMLs.

3.2 SPL basic analysis operations

Once abstract concepts have been defined, a number of basic operations can be properly specified, following the naming conventions used in [9]. They are summarized in Table 2.

Table 2 SPL basic analysis operations summary

Signature and Description	Motivation
$\_ \prec \_ : \mathit{Product} \leftrightarrow \mathit{SPL}$ Validity of a product with respect to an SPL	Determining whether a given product is available in an SPL
$\Pi : \mathit{SPL} \rightarrow \mathbb{F}\ \mathit{Product}$ The set of all valid products of an SPL	Identifying new requirements combinations not initially considered
$\mathcal{N} : \mathit{SPL} \rightarrow \mathbb{N}$ The number of all valid products of an SPL	Providing information about the flexibility and complexity of an SPL
$\mathit{void}\_ : \mathbb{P}\ \mathit{SPL}$ Void, i.e. empty, SPL	Debugging large–scale variability models
$\_ \prec_c \_ : \mathit{Configuration} \leftrightarrow \mathit{SPL}$ Validity of a configuration with respect to an SPL	Providing feedback during product design
$\Pi_\sigma : \mathit{SPL} \times \mathit{Configuration} \rightarrow \mathbb{F}\ \mathit{Product}$ SPL filtering	Selecting products according to key requirements

p_1 = {**Mobile Phone**, **Calls**, **Screen**, Basic}
p_2 = {**Mobile Phone**, **Calls**, **Screen**, Color}
p_3 = {**Mobile Phone**, **Calls**, **Screen**, Color, GPS}
p_4 = {**Mobile Phone**, **Calls**, **Screen**, High Resolution}
p_5 = {**Mobile Phone**, **Calls**, **Screen**, High Resolution, GPS}
p_6 = {**Mobile Phone**, **Calls**, **Screen**, Basic, Media, MP3}
p_7 = {**Mobile Phone**, **Calls**, **Screen**, Color, Media, MP3}
p_8 = {**Mobile Phone**, **Calls**, **Screen**, High Resolution, Media, Camera}
p_9 = {**Mobile Phone**, **Calls**, **Screen**, High Resolution, GPS, Media, Camera}
p_{10} = {**Mobile Phone**, **Calls**, **Screen**, Color, GPS, Media, MP3}
p_{11} = {**Mobile Phone**, **Calls**, **Screen**, High Resolution, Media, MP3}
p_{12} = {**Mobile Phone**, **Calls**, **Screen**, High Resolution, GPS, Media, MP3}
p_{13} = {**Mobile Phone**, **Calls**, **Screen**, High Resolution, Media, Camera, MP3}
p_{14} = {**Mobile Phone**, **Calls**, **Screen**, High Resolution, GPS, Media, Camera, MP3}

Fig. 4 Valid products of the SPL represented by the feature model in Figure 2

3.2.1 Validity of a product

As previously mentioned, a product is *valid* for an SPL (denoted as $\prec$) if it is configured using the SPL features and is an instance of its characteristic model. This operation may be helpful for SPL engineers to determine whether a given product is available in an SPL [9,75]. This can be expressed in Z as the following relation:

$$\_ \prec \_ : Product \leftrightarrow SPL$$

$$\forall\, p : Product;\ spl : SPL \bullet \\ p \prec spl \ \Leftrightarrow \\ (\ p \subseteq spl.features \ \land \ p \not\prec spl.model\ )$$

For example, consider the product p = {MobilePhone, Screen, Color, Media, MP3} and the SPL represented by the FM in Figure 2. Notice that p is not a valid product of the SPL because it does not include the mandatory feature Calls.

3.2.2 The set of all valid products

Using the validity relation ($\prec$), the set of all valid products of an SPL (denoted as Π), which may be helpful to identify new valid requirements combinations not considered in the initial scope of an SPL [9], can be defined in Z as the following function:

$$\Pi : SPL \rightarrow \mathbb{F}\, Product$$

$$\forall\, spl : SPL \bullet \\ \Pi\ spl\ =\ \{\ p : \mathbb{F}_1\ spl.features\ \mid\ p \prec spl\ \}$$

For example, in the case of the SPL represented by the FM in Figure 2, the set of all valid products is shown in Figure 4.

Notice that the computational complexity of the Π function as enunciated above is exponential with respect to the number of features of an SPL[3]. This is the reason why the Π function can be redefined in the CML of FLAME when it can be computed efficiently in a VML–dependent manner. Nevertheless, this is not a problem for automated testing, since previous works by some of the authors of this article [64] have shown that FMs with 10 features are complex enough to reveal faults effectively (see Section 5 for details).

[3] The size of the power set of a set S is 2 raised to the power of the number of elements in S.

3.2.3 The number of all valid products

Obviously, the number of valid products of an SPL (denoted as $\mathcal{N}$), which provides information about the flexibility and complexity of the SPL [9], is the cardinality (denoted in Z as $\#$) of its aforementioned set of products, i.e.:

$$\mathcal{N} : SPL \rightarrow \mathbb{N}$$

$$\forall\, spl : SPL \bullet \\ \mathcal{N}\ spl\ =\ \#\Pi\ spl$$

In the case of the SPL represented by the FM in Figure 2, the number of all valid products is 14 (see Figure 4).

3.2.4 Void SPL

An SPL is considered to be *void* if there does not exist any valid product for it. The automation of this operation is especially helpful when debugging large–scale feature models in which the manual detection of errors is recognized to be an error–prone and time–consuming task [9]. This can be expressed in Z by means of the following predicate:

$$void\_ : \mathbb{P}\, SPL$$

$$\forall\, spl : SPL \bullet \\ void\ spl\ \Leftrightarrow\ \Pi\ spl = \varnothing$$

As an example, consider the FMs in Figure 5, in which the introduction of a excludes constraint between the two mandatory features B and C in M makes the SPL represented by M' void.

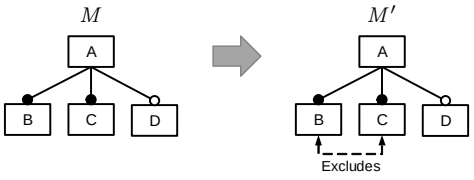


Fig. 5 Example of evolution of an SPL represented by a feature model into a void SPL

Theorem 1 (The number of products of void SPLs is 0)

$$\forall\, spl : SPL \bullet\ void\ spl\ \Leftrightarrow\ \mathcal{N}\ spl = 0$$

3.2.5 Validity of a configuration

As pointed out in [9], this operation is useful to provide feedback on the progress of a product configuration, i.e. an analysis tool implementing this operation could inform the user as soon as a configuration becomes invalid, thus saving time and effort.

Before specifying this operation, it necessary to define an auxiliary predicate similar to product validity for an SPL, i.e. a product is said to be valid with respect to a given configuration if all the selected features of the configuration are present in the product and none of the removed ones are. This can be specified in Z as follows:

$$\_ \lhd \_ : Product \leftrightarrow Configuration$$

$$\forall\, p : Product;\ c : Configuration \bullet$$
$$p \lhd c \ \Leftrightarrow$$
$$(\ selected\ c \subseteq p\ \land\ removed\ c \cap p = \varnothing\ )$$

Using the previous definition, a configuration is considered *valid* with respect to a given SPL if it is defined using *known* features and there exists at least one valid product in the SPL which is also *valid* for the given configuration. This validity concept can be specified as the following predicate:

$$\_ \prec_c \_ : Configuration \leftrightarrow SPL$$

$$\forall\, c : Configuration;\ spl : SPL \bullet$$
$$c \prec_c spl \ \Leftrightarrow$$
$$(\ selected\ c \cup removed\ c\ ) \subseteq spl.features$$
$$\land\ \exists\, p : \Pi\ spl \bullet\ p \lhd c$$

Theorem 2 (There not exists any valid configuration for a void SPL)

$$\forall\, spl : SPL \bullet$$
$$void\ spl \ \Rightarrow\ \nexists\, c : Configuration \bullet\ c \prec_c spl$$

3.2.6 SPL filtering

A *filtering* or product selection of an SPL over a given configuration is the set of products of the SPL which are valid for the given configuration. As commented in [9], this operation may be helpful to assist users during the configuration process. Firstly, they can filter the products according to their key requirements. Then, the list of resultant products can be inspected to select the desired solution. The specification of this operation in Z is as follows:

$$\Pi_\sigma : SPL \times Configuration \rightarrow \mathbb{F}\ Product$$

$$\forall\, spl : SPL;\ c : Configuration \bullet$$
$$\Pi_\sigma(\ spl, c\ ) = \{\ p : \Pi\ spl \mid p \lhd c\ \}$$

For example, if the set of products in Figure 4 is filtered using the partial configuration ({Calls, GPS}, {Color, Camera}), the result is the set of products composed by p_5 = {Mobile Phone, Calls, GPS, Screen, High Resolution} and p_{12} = {Mobile Phone, Calls, GPS, Screen, High Resolution, Media, MP3}, which are the only two valid products with respect to the configuration.

Theorem 3 (Any filtering on a void SPL results in an empty set of products)

$$\forall\, spl : SPL;\ c : Configuration \bullet$$
$$void\ spl \ \Rightarrow\ \Pi_\sigma(\ spl, c\ ) = \varnothing$$

3.3 SPL relations

As most software engineering artifacts, SPLs evolve during their lifecycle. In [9], a number of relations between SPLs that can provide interesting feedback during SPL evolution are described. They are summarized in Table 3.

3.3.1 SPL equivalence (refactoring)

An SPL is considered as a *refactoring* of another SPL if they both represent the same set of valid products, although their sets of features and characteristic models respectively do not have to be the same. In this case, they are also said to be *equivalent*. This can be expressed in Z as follows:

$$\_ \equiv \_ : SPL \leftrightarrow SPL$$

$$\forall\, spl_1, spl_2 : SPL \bullet$$
$$spl_1 \equiv spl_2 \ \Leftrightarrow\ \Pi\ spl_1 = \Pi\ spl_2$$

For example, the SPLs represented by M and M_1 in Figure 6 are equivalent because their sets of valid products are the same although their models are different.

Table 3 SPL relations summary

Signature and Description	Motivation
$\_ \equiv \_ : SPL \leftrightarrow SPL$	SPL evolution feedback
SPL equivalence, same set of products	
$\_ \sqsubset \_ : SPL \leftrightarrow SPL$	Same as above
SPL specialization, product subset	
$\_ \sqsupset \_ : SPL \leftrightarrow SPL$	Same as above
SPL generalization, product superset	
$\_ \langle\langle\rangle\rangle \_ : SPL \leftrightarrow SPL$	Same as above
SPL arbitrary edit, none of the above	

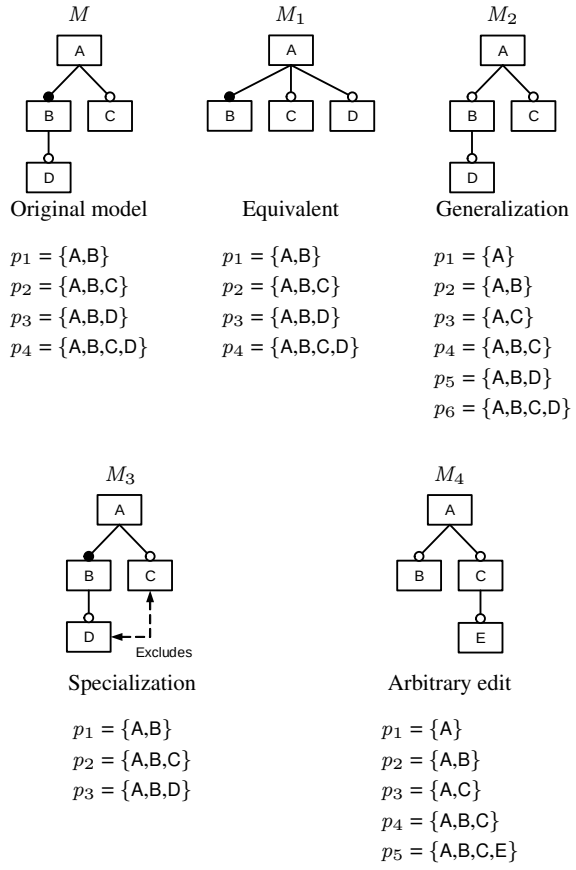


Fig. 6 Examples of relationships between SPLs represented by feature models

Theorem 4 (Any pair of void SPLs are equivalent)

$$\forall\, spl_1, spl_2 : SPL \bullet \\ (\ void\ spl_1\ \land void\ spl_2\ )\ \Rightarrow\ spl_1 \equiv spl_2$$

3.3.2 SPL generalization/specialization

An SPL is considered as a *generalization* of another SPL if its set of valid products is a superset of the products of the latter SPL. Inversely, an SPL is considered as a *specialization* of another SPL if its set of valid products is a subset of the latter SPL. Both relations can be expressed in Z as the following:

$$\_ \sqsubset \_ : SPL \leftrightarrow SPL$$
$$\_ \sqsupset \_ : SPL \leftrightarrow SPL$$
$$\forall\, spl_1, spl_2 : SPL \bullet \\ (\ spl_1 \sqsubset\ spl_2\ \Leftrightarrow\ \Pi\ spl_1 \subset \Pi\ spl_2\ ) \land \\ (\ spl_2 \sqsupset\ spl_1\ \Leftrightarrow\ spl_1 \sqsubset\ spl_2\ )$$

For example, the SPL represented by M_2 in Figure 6 is a generalization of the SPL represented by M because its set of valid products is the same as of M plus the {A} and {A,C} products. On the other hand, the SPL represented by M_3 is an specialization of the SPL represented by M because its set of products is the same as of M minus the {A,B,C,D} product.

3.3.3 SPL arbitrary edit

The last SPL evolution relation is the so–called *arbitrary edit*, which is the kind of relation between two SPLs when they are neither equivalent nor a generalization or specialization of each other (see [67] for details). This can be expressed in Z as the following relation:

$$\_\langle\langle\rangle\rangle\_ : SPL \leftrightarrow SPL$$
$$\forall\, spl_1, spl_2 : SPL \bullet \\ spl_1\ \langle\langle\rangle\rangle\ spl_2\ \Leftrightarrow \\ \neg\ (\ spl_1\ \equiv\ spl_2\ ) \land \quad \text{[not refactoring]} \\ \neg\ (\ spl_1\ \sqsubset\ spl_2\ ) \land \quad \text{[not specialization]} \\ \neg\ (\ spl_1\ \sqsupset\ spl_2\ ) \quad \text{[not generalization]}$$

For example, the SPL represented by M_4 in Figure 6 is an arbitrary edit of the SPL represented by M because their sets of valid products, although not disjoint, are neither the same nor a subset of each other.

3.4 SPL feature–related operations

The following operations provide the SPL engineer with relevant information about the presence or absence of the features of a given SPL in its set of valid products that can lead to changes in the corresponding characteristic model. They are summarized in Table 4.

3.4.1 Core features

The *core* features of an SPL (denoted as Φ_C) are those features that appear in all products of the SPL. As commented in [9], this operation is useful to determine which features should be developed in first place or to decide which features should be part of the core architecture of the SPL. This concept can be easily expressed in Z by means of the *generalized intersection*[4] (denoted as $\bigcap$) operator over the set of valid products:

$$\Phi_C : SPL \rightarrow \mathbb{F}\ Feature$$
$$\forall\, spl : SPL \bullet \\ \Phi_C\ spl\ =\ \bigcap\ \Pi\ spl$$

[4] The generalized intersection over A, being A a set of sets, is the set consisting of all objects belonging to every set in A.

Table 4 SPL feature–related operations summary

Signature and Description	Motivation
$\Phi_C : SPL \rightarrow \mathbb{F}\ Feature$ Core features, present in all products	Architectural design, development prioritization
$\Phi_D : SPL \rightarrow \mathbb{F}\ Feature$ Dead features, not present in any product	Detecting inconsistencies in SPL design
$\Phi_V : SPL \rightarrow \mathbb{F}\ Feature$ Variant features, present in some products	Architectural design, development prioritization
$\Phi_U : SPL \rightarrow \mathbb{F}\ Feature$ Unique features, present in only one product	Architecture design, development prioritization
$\Phi_A : SPL \rightarrow \mathbb{F}\,\mathbb{F}_1\ Feature$ Atomic sets of features which always appear together	Detection of feature coupling, efficient preprocessing

For example, the set of core features of the SPL represented by the FM in Figure 2, as can be deduced from its list of valid products in Figure 4 where they are displayed in bold face, is {Mobile Phone, Calls, Screen}.

Theorem 5 (The set of core features of a void SPL is empty)

$$\forall\, spl : SPL \bullet\ void\ spl \Rightarrow \Phi_C\ spl = \varnothing$$

3.4.2 Dead features

On the other hand, features that do not appear in any product of their SPL are said to be *dead* features, which are undesired inconsistencies whose detection is essential in SPL engineering. This can be expressed in Z by means of the set difference between the SPL features and the *generalized union*[5] (denoted as $\bigcup$) over the set of products, i.e. all the features appearing in at least one valid product.

$$\Phi_D : SPL \rightarrow \mathbb{F}\ Feature$$
$$\forall\, spl : SPL \bullet$$
$$\Phi_D\ spl = spl.features \setminus \bigcup \Pi\ spl$$

For example, all features in gray in Figure 7 are dead features of the SPLs represented by the corresponding FMs. Notice that in FMs, dead features are caused by a wrong usage of cross–tree constraints [9].

[5] The generalized union over A, being A a set of sets, is the set consisting of all objects belonging to any set in A.

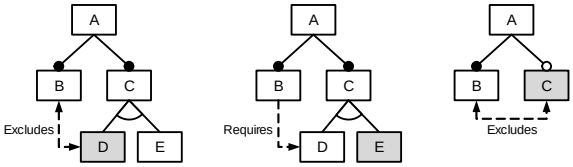


Fig. 7 Examples of dead features (in gray) in SPLs represented by feature models

Theorem 6 (All features of a void SPL are dead)

$$\forall\, spl : SPL \bullet\ void\ spl \Rightarrow \Phi_D\ spl = spl.features$$

3.4.3 Variant features

The *variant* features of an SPL are those features that appear only in some products of the SPL, i.e. the features that are neither *core* features nor *dead* features, something that can be easily expressed in Z by means of the set difference between the SPL features and its core and dead features:

$$\Phi_V : SPL \rightarrow \mathbb{F}\ Feature$$
$$\forall\, spl : SPL \bullet$$
$$\Phi_V\ spl = spl.features \setminus \Phi_C\ spl \setminus \Phi_D\ spl$$

For example, the set of variant features of the SPL represented by the FM in Figure 2, as can be deduced from its list of valid products in Figure 4, is {GPS, Basic, Color, High Resolution, Media, Camera, MP3}. Notice that, as a result of the work described in this article, the definition of this operation differs from the presented in [9] (see Section 5.3.1).

Theorem 7 (The set of variant features of a void SPL is empty)

$$\forall\, spl : SPL \bullet\ void\ spl \Rightarrow \Phi_V\ spl = \varnothing$$

Theorem 8 (The core, variant, and dead features of an SPL partition its features)

$$\forall\, spl : SPL \bullet$$
$$\langle\, \Phi_C\ spl,\ \Phi_V\ spl,\ \Phi_D\ spl \,\rangle\ partitions\ spl.features$$

3.4.4 Unique features

Those features that appear in only one valid product are said to be *unique*, and are used to measure the *homogeneity* of an SPL (see Section 3.5.3). This operation can be specified in Z using the *unique existential quantifier* ($\exists_1$):

$$\Phi_U : SPL \rightarrow \mathbb{F}\ Feature$$
$$\forall\, spl : SPL \bullet$$
$$\Phi_U\ spl = \{\, f_u : spl.features \mid \exists_1\, p : \Pi\ spl \bullet f_u \in p \,\}$$

Theorem 9 (The set of unique features of a void SPL is empty)

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \Phi_U\ spl = \varnothing$$

Theorem 10 (In SPLs with more than one product, unique features are variant features)

$$\forall\, spl : SPL \bullet \mathcal{N}\ spl > 1 \Rightarrow \Phi_U\ spl \subseteq \Phi_V\ spl$$

Theorem 11 (In SPLs with only one product, unique features are core features)

$$\forall\, spl : SPL \bullet \mathcal{N}\ spl = 1 \Rightarrow \Phi_U\ spl = \Phi_C\ spl$$

3.4.5 Atomic sets of features

First mentioned in [79] but not formalized yet, the concept of the *atomic sets* of features of an SPL is relevant as an efficient preprocessing technique for SPL automated analysis [9,60]. Informally, an atomic set is a group of features that can be treated as a unit because they are tightly coupled and always appear together in the SPL products. Atomic sets can be used to create a reduced version of the SPL characteristic model simply by replacing groups of features with the atomic set containing them [9], thus increasing the efficiency of other SPL analysis operations.

From a formal point of view, atomic sets are nonempty subsets of features such that for every product in an SPL, all their features appear together in the product or none of them appear at all, i.e. they are a *subset or disjoint* with respect to every product. There are many atomic sets for a given SPL, but the interesting ones are the *maximal* subsets which are not contained in any other atomic set and in which the features are grouped in the biggest groups.

In order to make the formal specification of the atomic sets easier to understand, the set of all *potential* atomic sets (denoted as Φ_A^0) is defined first. This set contains all feature subsets with *subset–or–disjoint* semantics:

$$\Phi_A^0 : SPL \rightarrow \mathbb{F}\,\mathbb{F}_1\ Feature$$

$$\forall\, spl : SPL \bullet \Phi_A^0\ spl = \{\ a_0 : \mathbb{F}_1\ spl.features \mid \forall\, p : \Pi\ spl \bullet a_0 \subseteq p \ \vee\ a_0 \cap p = \varnothing\ \}$$

Once Φ_A^0 is defined, the maximal set of atomic sets (denoted as Φ_A) is defined as the set of those atomic sets which are not included in any other atomic set. Although the Z notation does not include a $maximal$ relation, it can be easily defined as a generic relation as follows:

$$[X]$$

$$maximal : \mathbb{P}\,X \leftrightarrow \mathbb{P}\,\mathbb{P}\,X$$

$$\forall\, S_i : \mathbb{P}\,X;\ S : \mathbb{P}\,\mathbb{P}\,X \bullet maximal(\ S_i, S\ ) \Leftrightarrow S_i \in S\ \wedge\ \nexists\, S_j : S \bullet S_i \subset S_j$$

Now, having defined the $maximal$ relation, Φ_A can be specified as those potential atomic sets that are *maximal*, i.e.:

$$\Phi_A : SPL \rightarrow \mathbb{F}\,\mathbb{F}_1\ Feature$$

$$\forall\, spl : SPL \bullet \Phi_A\ spl = \{\ a : \Phi_A^0\ spl \mid maximal(\ a,\ \Phi_A^0\ spl\ )\ \}$$

For example, the atomic sets of the SPL represented by the FM in Figure 2, as can be deduced from its list of valid products in Figure 4, are { {Mobile Phone, Calls, Screen}, {GPS}, {Basic}, {Color}, {High Resolution}, {Media}, {Camera}, {MP3} }.

Notice that the definition of this operation differs significatively from previous informal definitions provided in [9], [60] and [79], which are VML–dependent whereas the one provided in this article is much more abstract and independent from the notation used for the characteristic model of an SPL. More details are provided in Section 5.3.3.

Theorem 12 (The core features, if any, are always one of the atomic sets)

$$\forall\, spl : SPL \bullet \Phi_C\ spl \neq \varnothing \Rightarrow \Phi_C\ spl \in \Phi_A\ spl$$

Theorem 13 (The dead features, if any, are always one of the atomic sets)

$$\forall\, spl : SPL \bullet \Phi_D\ spl \neq \varnothing \Rightarrow \Phi_D\ spl \in \Phi_A\ spl$$

Theorem 14 (Void SPLs only have one atomic set, its features)

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \Phi_A\ spl = \{\ spl.features\ \}$$

3.5 SPL numerical indicators

Apart from the number of products ($\mathcal{N}$) defined in Section 3.2.3, other VML–independent numerical indicators are defined in this section and summarized in Table 5. As a result of the test–based validation of the FLAME framework, the definitions of some indicators have been enhanced with respect to the presented in [9] in order to correct some mistakes and to avoid division by zero in some quotients. See Section 5.3.2 for details.

Table 5 SPL numerical indicators

Signature and Description	Motivation
$\mathcal{C} : SPL \times Configuration \to \mathbb{R}$ Commonality factor of configuration	Architectural design, development prioritization
$\mathcal{V} : SPL \to \mathbb{R}$, $\mathcal{V}_\rho : SPL \to \mathbb{R}$ SPL total & partial variability	Metric for SPL design
$\mathcal{H} : SPL \to \mathbb{R}$, $\mathcal{H}_{old} : SPL \to \mathbb{R}$ SPL homogeneity (new & old definitions)	Metric for SPL design

3.5.1 Commonality factor of a configuration

The *commonality factor* of a configuration in an SPL (denoted as $\mathcal{C}$) is the percentage of products of the SPL including the given configuration (0 if the SPL is void). Like the previously specified *core features* (see Section 3.4.1), this operation may be used to prioritize the development order of the features or to decide which features should be part of the core architecture of the SPL [9]. Its specification in Z is as follows:

$$\mathcal{C} : SPL \times Configuration \to \mathbb{R}$$

$$\forall\, spl : SPL;\ c : Configuration \bullet$$
$$void\ spl \Rightarrow \mathcal{C}(\ spl, c\ ) = 0$$
$$\forall\, spl : SPL;\ c : Configuration \bullet$$
$$\neg\ void\ spl \Rightarrow$$
$$\mathcal{C}(\ spl, c\ ) = \#\Pi_\sigma(\ spl, c\ ) \,/\, \mathcal{N}\ spl$$

For example, the percentage of products of the SPL represented by the FM in Figure 2 able to play MP3 files but without GPS is given by the commonality of the partial configuration ({MP3}, {GPS}), which is 4/14 = 28.57%, as can be deduced from its list of valid products in Figure 4.

3.5.2 SPL variability

The *variability* of an SPL, considered as a measure of its flexibility, is defined in [9] as the ratio between the number of its valid products and the number of the potential products it could have, i.e. $2^n - 1$ where n is the number of features under consideration. If all the SPL features are considered, the variability is referred to as *total variability* ($\mathcal{V}$) whereas if only *variant features* (see Section 3.4.3) are considered, it is referred to as *partial variability* ($\mathcal{V}_\rho$), which is 0 in case the SPL has no variant features, i.e. in the case the SPL is void (see Theorem 7) or has only one valid product (see Theorem 11). Their Z specifications are the following:

$$\mathcal{V} : SPL \to \mathbb{R}$$
$$\mathcal{V}_\rho : SPL \to \mathbb{R}$$

$$\forall\, spl : SPL \bullet$$
$$\mathcal{V}\ spl = \frac{\mathcal{N}\ spl}{2^{\#spl.features} - 1} \quad \wedge$$
$$\Phi_V\ spl = \varnothing \Rightarrow \mathcal{V}_\rho\ spl = 0 \quad \wedge$$
$$\Phi_V\ spl \neq \varnothing \Rightarrow \mathcal{V}_\rho\ spl = \frac{\mathcal{N}\ spl}{2^{\#(\Phi_V\ spl)} - 1}$$

For example, for the SPL represented by the FM in Figure 2, its total variability is 14/($2^{10} - 1$) = 1.37%, whereas its partial variability is 14/($2^7 - 1$) = 11.02%.

Theorem 15 (The total variability of a void SPL is 0)

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \mathcal{V}\ spl = 0$$

Theorem 16 (The partial variability of a void SPL is 0)

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \mathcal{V}_\rho\ spl = 0$$

3.5.3 SPL homogeneity

According to [26,27] —but not exactly to [9], see Section 5.3.2 for details— the *homogeneity* of an SPL is related to the number of their *unique features* (see Section 3.4.4). The more unique features an SPL has, the less homogeneous the SPL is. Formally, the homogeneity of an SPL was initially described in [26] as a percentage defined as one minus the ratio between the number of unique features and the number of features, i.e. an SPL without unique features would have an homogeneity of 100% whereas another one with a 25% of unique features would have an homogeneity of 75%. This can be expressed in Z as follows:

$$\mathcal{H}_{old} : SPL \to \mathbb{R}$$

$$\forall\, spl : SPL \bullet$$
$$\mathcal{H}_{old}\ spl = 1 - \frac{\#(\Phi_U\ spl)}{\#spl.features}$$

Recently, the definition of this indicator has been revised in [27] due to erroneous results in some scenarios. For example, as commented in [27], consider an SPL with 200 products and 50 features, in which every feature is included in just two products; although the SPL is clearly quite heterogeneous, the $\mathcal{H}_{old}$ indicator would say that the SPL is totally homogeneous. Consider also a void SPL, for which the value of $\mathcal{H}_{old}$ would be 100%, something which is at least debatable.

In order to avoid these problems, the homogeneity of an SPL has been redefined in [27] as the commonality mean, i.e. the sum of the commonality factor of all the features[6]

[6] The commonality factor of a single feature f is the commonality factor of a configuration with the single feature f as selected and no removed features, i.e. $\mathcal{C}(spl, (\{f\}, \varnothing))$.

in the SPL divided by the number of features. This can be expressed in Z as follows:[7]

$$\mathcal{H} : SPL \to \mathbb{R}$$

$$\forall\, spl : SPL \bullet \mathcal{H}\ spl = \frac{\sum \mathcal{C}(\ spl,\ (\{f_i\}, \varnothing)\ )}{\#spl.features} \quad f_i \in spl.features$$

For example, for the SPL represented by the FM in Figure 2, which has no unique features, its homogeneity is 100% according to the former definition, whereas it is 58.57% according to the latter definition. According to its list of valid products in Figure 4, the latter homogeneity value seems much more appropriate.

Theorem 17 (The old homogeneity of a void SPL is 100%)

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \mathcal{H}_{old}\ spl = 100\%$$

Theorem 18 (The new homogeneity of a void SPL is 0%)

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \mathcal{H}\ spl = 0\%$$

4 Characteristic model layer of the FLAME framework

The *characteristic model layer* (CML) is the layer of FLAME in which the specific aspects of different VMLs are taken into consideration. As mentioned in Section 3.1, at least the abstract type $Model$, and the abstract operations Φ (*features–in–a–model*) and $\nprec$ (*is–instance–of*) have to be specified in order to formalize an VML.

Among the different VMLs, the FODA–like *basic feature model* (BFM) notation, as described in [9], has been chosen for its formalization for being one of the most widely used in the SPL community. According to this decision, the *abstract syntax* of BFM and the abstract operations declared in the AFL are specified in the rest of this section.

4.1 BFM as a characteristic model

Following [9], a BFM is a characteristic model in which features are organized hierarchically using *mandatory*, *optional*, *only–one* or *one–or–more* relationships. A BFM can also include the so–called *cross–tree constraints* (CTCs), which can express (1) that a feature *requires* the presence of another feature; or (2) that a feature *excludes* another feature, i.e. that they are incompatible and therefore cannot appear together in a product. All FMs used in the figures in this article use BFM as its notation.

[7] The use of the *summation symbol* (Σ) over the elements of a set is not explicitly defined in Z, but we have decided to use it for the sake of understandability.

4.1.1 BFM metamodel

The first task to formalize a VML in FLAME is to specify the abstract type $Model$, which usually represents the VML metamodel. For that purpose, the metamodel in Figure 8 was first developed in UML, then formalized in Z by means of the so–called *free types* [65], which are usually used to specify *abstract syntaxes*. Finally, the abstract syntax was translated into Prolog functors (see Appendix D for details).

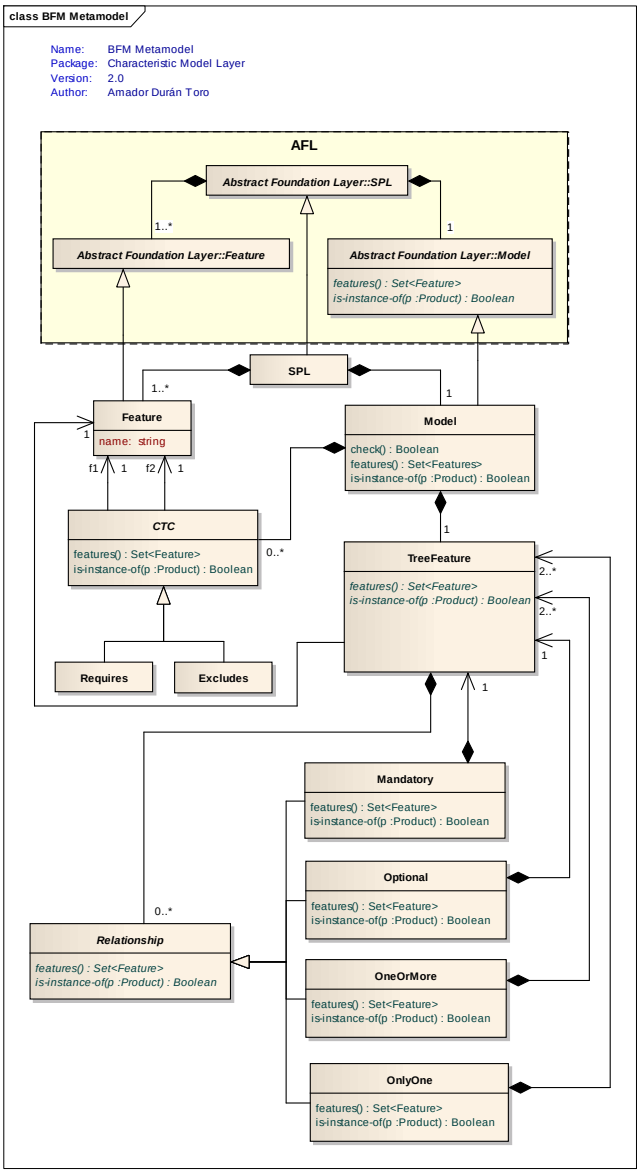


Fig. 8 BFM metamodel

Following this approach, the main symbol in the BFM abstract syntax is used to redefine the abstract type $Model$ as a pair formed by a *feature tree* and a finite set of *cross–tree–constraints* which could be empty:

$$Model\ ::=\ BFM\ \langle\!\langle\ FeatureTree \times \mathbb{F}\ CTC\ \rangle\!\rangle$$

Then, feature trees are defined as having a feature name and a set, which could be empty, of *mandatory*, *optional*, *one–or–more* or *only–one* relationships:

$$
\begin{array}{rcl}
FeatureTree & ::= & feature \langle\!\langle\ Feature \times \mathbb{F}\ Relationship\ \rangle\!\rangle \\
Relationship & ::= & mandatory\ \langle\!\langle\ FeatureTree\ \rangle\!\rangle \\
& | & optional\ \langle\!\langle\ FeatureTree\ \rangle\!\rangle \\
& | & one\_or\_more\ \langle\!\langle\ \mathbb{F}_2\ FeatureTree\ \rangle\!\rangle \\
& | & only\_one\ \langle\!\langle\ \mathbb{F}_2\ FeatureTree\ \rangle\!\rangle
\end{array}
$$

Notice that the *one–or–more* and *only–one* relationships are compound of a finite set of at least two feature trees, as indicated by the corresponding multiplicities in the BFM metamodel in Figure 8. In this case, $\mathbb{F}_2$ is generically defined as $\mathbb{F}_2\ X == \{\ S : \mathbb{F}\ X\ \mid\ \#S \geq 2\ \}$, i.e. the type of finite sets with at least two elements.

The other elements in BFMs are the cross–tree–constraints, which are defined as the CTC type formed by pairs of feature names which *require* or *exclude* each other:[8]

$$
\begin{array}{rcl}
CTC & ::= & requires\ \langle\!\langle\ Feature \times Feature\ \rangle\!\rangle \\
& | & excludes\ \langle\!\langle\ Feature \times Feature\ \rangle\!\rangle
\end{array}
$$

As an example, the BFM in figure 2 can be represented in the previously defined abstract syntax as follows:

$$
\begin{array}{l}
BFM\ (\\
\quad feature(\ MobilePhone, \{ \\
\quad\quad mandatory(\ feature(Calls, \varnothing)\ ), \\
\quad\quad optional(\ feature(GPS, \varnothing)\ ), \\
\quad\quad mandatory(\ feature(Screen), \{ \\
\quad\quad\quad only\_one(\ \{ \\
\quad\quad\quad\quad feature(Basic, \varnothing), \\
\quad\quad\quad\quad feature(Color, \varnothing), \\
\quad\quad\quad\quad feature(HighResolution, \varnothing) \\
\quad\quad\quad\}\ ) \\
\quad\quad\}\ ) \\
\quad\quad optional(\ feature(Media), \{ \\
\quad\quad\quad one\_or\_more(\ \{ \\
\quad\quad\quad\quad feature(Camera, \varnothing), \\
\quad\quad\quad\quad feature(MP3, \varnothing) \\
\quad\quad\quad\}\ ) \\
\quad\quad\}\ ) \\
\quad\}\ ), \\
\quad\{ \\
\quad\quad excludes(GPS, Basic), \\
\quad\quad requires(Camera, HighResolution) \\
\quad\} \\
)
\end{array}
$$

[8] Other approaches like [5] propose the use of propositional logic, e.g. *well–formed–formulas*, for CTCs. See for example [7], in which a very preliminary version of this work includes them.

4.1.2 Helper functions for BFM specification

In order to make the specification easier to read, some helper functions can be defined over the previously defined BFM abstract syntax. The first ones are simply a pair of functions used to extract the feature tree, and the set of CTCs from a given BFM:

$$
\begin{array}{|l}
tree : Model \rightarrow FeatureTree \\
ctc : Model \rightarrow \mathbb{F}\ CTC \\
\hline
\forall\ t : FeatureTree;\ c : \mathbb{F}\ CTC \bullet \\
\quad tree\ BFM(t, c)\ =\ t\ \land \\
\quad ctc\ BFM(t, c)\ =\ c
\end{array}
$$

Another helper function is the *children* function, which returns the set of children feature subtrees in a relationship. Its specification in Z is as follows:

$$
\begin{array}{|l}
children : Relationship \rightarrow \mathbb{F}_1\ FeatureTree \\
\hline
\forall\ t_i : FeatureTree;\ t : \mathbb{F}_2\ FeatureTree \bullet \\
\quad children\ mandatory(\ t_i\ ) = \{\ t_i\ \}\ \land \\
\quad children\ optional(\ t_i\ ) = \{\ t_i\ \}\ \ \land \\
\quad children\ one\_or\_more(\ t\ ) = t\ \ \land \\
\quad children\ only\_one(\ t\ ) = t
\end{array}
$$

The last helper functions are φ_τ, φ_R, and φ_χ, which respectively return the *bag* of feature names used in a *FeatureTree*, in a *Relationship*, and in a CTC. In Z, bags (also known as *multisets*) are represented as lists of unordered elements allowing duplicates, i.e. $[\![a, b, b, c, d]\!]$. Formally, a bag of objects of type T is a partial function $T \nrightarrow \mathbb{N}$, i.e. a function that assigns each object in the bag the number of times it appears in the bag. For example, the former sample bag can also be represented as the function extension $\{a \mapsto 1, b \mapsto 2, c \mapsto 1, d \mapsto 1\}$. As in all functions, the *domain* (dom) and *range* (ran) operators can be applied returning respectively the set of objects in the bag (without duplicates) and the set of cardinalities (also without duplicates), i.e. dom $[\![a, b, b, c, d]\!] = \{a, b, c, d\}$ whereas ran $[\![a, b, b, c, d]\!] = \{1, 2\}$.

Having said that, the φ_τ, φ_R, and φ_χ functions can be defined as follows, in which the $\uplus$ and $\biguplus$ symbols represent respectively the bag union and the *generalized bag union*[9] over a bag of bags:

[9] The generalized bag union over A, being A a bag of bags, is the bag consisting on the union of all the bags in A. Although it is not explicitly defined in Z, we have decided to use it for the sake of understandability.

$$\varphi_\tau : \mathit{FeatureTree} \rightarrow \text{bag}\ \mathit{Feature}$$
$$\varphi_R : \mathit{Relationship} \rightarrow \text{bag}\ \mathit{Feature}$$
$$\varphi_\chi : \mathit{CTC} \rightarrow \text{bag}\ \mathit{Feature}$$

$$\begin{aligned}
&\forall\ f_1, f_2 : \mathit{Feature};\ r_i : \mathit{Relationship};\\
&\quad r : \mathbb{F}\ \mathit{Relationship} \bullet\\
&\qquad \varphi_\tau\ \mathit{feature}(\ f_1, r\ ) \ =\ [\![\ f_1\ ]\!] \uplus\\
&\qquad\qquad \biguplus [\![\ r_j : r \bullet \varphi_R\ r_j\ ]\!] \ \land\\
&\qquad \varphi_R\ r_i \ =\ \biguplus [\![\ t_j : \mathit{children}\ r_i \bullet \varphi_\tau\ t_j\ ]\!] \ \land\\
&\qquad \varphi_\chi\ \mathit{requires}(\ f_1, f_2\ ) \ =\ [\![\ f_1, f_2\ ]\!] \ \land\\
&\qquad \varphi_\chi\ \mathit{excludes}(\ f_1, f_2\ ) \ =\ [\![\ f_1, f_2\ ]\!]
\end{aligned}$$

In other words, the bag of features of a feature tree is formed by its root feature and the union of all the feature bags from its relationships. In turn, the bag of features of a relationship is the union of all the feature bags from its children subtrees. Finally, the bag of features of a CTC is formed by the two features involved in the constraint.

4.2 Redefining the *features–in–a–model* function

The Φ function must also be redefined in order to make concrete the abstract definitions in the AFL. First, the feature names in a BFM are specified as the union of the feature names in its feature tree (Φ_τ) and in all its CTCs (Φ_χ). Then, the Φ_τ and Φ_χ functions are defined as the domains of the corresponding feature bags φ_τ and φ_χ defined in the previous section, i.e.:

$$\Phi : \mathit{Model} \rightarrow \mathbb{F}\ \mathit{Feature}$$
$$\Phi_\tau : \mathit{FeatureTree} \rightarrow \mathbb{F}\ \mathit{Feature}$$
$$\Phi_\chi : \mathit{CTC} \rightarrow \mathbb{F}\ \mathit{Feature}$$

$$\begin{aligned}
&\forall\ m : \mathit{Model};\ t_i : \mathit{FeatureTree};\ \mathit{ctc}_i : \mathit{CTC} \bullet\\
&\quad \Phi\ m \ =\ \Phi_\tau\ \mathit{tree}\ m \cup\\
&\qquad\qquad \bigcup \{\ \mathit{ctc}_j : \mathit{ctc}\ m \bullet \Phi_\chi\ \mathit{ctc}_j\ \} \ \land\\
&\quad \Phi_\tau\ t_i \ =\ \mathrm{dom}(\ \varphi_\tau\ t_i\ ) \ \land\\
&\quad \Phi_\chi\ \mathit{ctc}_i \ =\ \mathrm{dom}(\ \varphi_\chi\ \mathit{ctc}_i\ )
\end{aligned}$$

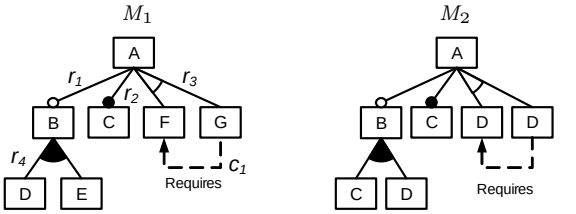
For example, in the case of the BFM in Figure 2, its set of feature names would be: dom [[Mobile Phone, Calls, GPS, Screen, Basic, Color, High Resolution, Media, Camera, MP3]] $\cup$ dom [[GPS, Basic]] $\cup$ dom [[Camera, High Resolution]] $=$ { Mobile Phone, Calls, GPS, Screen, Basic, Color, High Resolution, Media, Camera, MP3 }.

4.3 Redefining the *SPL* type

As commented at the end of section 3.1.1, the abstract SPL schema type can be augmented with additional invariants related to the nature of features or models. In the case of BFMs, two invariants apart from the one defined in the AFL have to be added in order to guarantee the structural correctness of the models, resulting in the following schema type:

$$\begin{aligned}
&SPL\\
&\mathit{model} : \mathit{Model} \qquad \text{[SPL using BFM]}\\
&\mathit{features} : \mathbb{F}_1\ \mathit{Feature}\\
&\Phi\ \mathit{model} = \mathit{features}\\
&\mathrm{ran}(\ \varphi_\tau\ \mathit{tree}\ \mathit{model}\ ) \ =\ \{\ 1\ \}\\
&\forall\ \mathit{ctc}_i : \mathit{ctc}\ \mathit{model} \bullet\\
&\quad \Phi_\chi\ \mathit{ctc}_i \ \subseteq\ \Phi_\tau\ \mathit{tree}\ \mathit{model}
\end{aligned}$$

The first invariant, considering that BFMs are structurally trees and that feature names are usually unique identifiers in FODA–like VMLs, states that the same feature name cannot appear more than once in the SPL feature tree. For example, in the case of M_1 in Figure 9, the bag of feature names of its feature tree is [[A, B, C, D, E, F, G]], in which all feature names appear only once, i.e. the bag range is $\{1\}$. On the other hand, the corresponding bag for M_2 is [[A, B, C, C, D, D, D]], whose range is $\{1, 2, 3\}$ because two feature names (C and D) appear more than once thus making the SPL represented by M_2 incorrect.



a) Structurally correct BFM b) Structurally incorrect BFM

Fig. 9 Examples of correct and incorrect BFMs

The second invariant states that the feature names in the CTCs must be a subset of the feature names in the feature tree, i.e. that all CTCs must be defined over features appearing in the feature tree. For example, a CTC such as G requires H in M_1 in Figure 9 would be incorrect because H does not appear in the M_1 feature tree.

4.4 Redefining the *is–instance–of* relation

The redefinition of the $\prec\!\!\!<$ relation is structurally similar to the redefinition of the Φ function. In this case, the first step is to specify whether a product is an instance of an SPL using a BFM as its characteristic model. Basically, a product is an instance of such an SPL if it is an instance of its feature tree and of all their CTCs, i.e.:

$$\_ \prec \_ : Product \leftrightarrow Model$$

$$\forall\, p : Product;\ m : Model \bullet$$
$$p \prec m \ \Leftrightarrow\ (\ p \prec_\tau tree\ m\ \wedge$$
$$\forall\, ctc_i : ctc\ m \bullet\ p \prec_\chi ctc_i\ )$$

The second step is the specification of the $\prec$ relation for BFM trees (denoted as $\prec_\tau$) and for its relationships (denoted as $\prec_R$). In the former relation, a product is an instance of a feature tree if it includes the parent feature name and is an instance of all its children relationships, i.e.:

$$\_ \prec_\tau \_ : Product \leftrightarrow FeatureTree$$

$$\forall\, p : Product;\ f : Feature;\ r : \mathbb{F}\, Relationship \bullet$$
$$p \prec_\tau feature(\ f,\ r\ ) \ \Leftrightarrow$$
$$(\ f \in p\ \wedge\ \forall\ r_i : r \bullet\ p \prec_R r_i\ )$$

For example, a product is an instance of the A feature tree in Figure 9(a) if it contains feature A and it is also an instance of relationships r_1, r_2, and r_3.

With respect to relationships, four cases have to be considered. In the case of *mandatory* subtrees, a product is an instance if is an instance of the mandatory subtree, i.e.:

$$\_ \prec_R \_ : Product \leftrightarrow Relationship$$

$$\forall\, p : Product;\ t_i : FeatureTree \bullet$$
$$p \prec_R mandatory(\ t_i\ ) \ \Leftrightarrow\ p \prec_\tau t_i$$

For example, in Figure 9(a) a product is an instance of the mandatory relationship r_2 if it is an instance of the C subtree, i.e. if it contains C.

In the case of *optional* subtrees, a product is an instance if it is an instance of the optional subtree or it is disjoint from the subtree features. Formally:

$$\_ \prec_R \_ : Product \leftrightarrow Relationship$$

$$\forall\, p : Product;\ t_i : FeatureTree \bullet$$
$$p \prec_R optional(\ t_i\ ) \ \Leftrightarrow$$
$$(\ p \prec_\tau t_i\ \vee\ p \cap \Phi_\tau\ t_i\ =\ \varnothing\ )$$

For example, in Figure 9(a) a product is an instance of the optional relationship r_1 if (1) it is an instance of the B subtree, i.e. it contains {B, D}, or {B, E}, or {B, D, E} or, (2) it is disjoint from the subtree features, i.e. it does not contain any of {B, D, E}.

In the case of *one_or_more* and *only_one* subtrees, all their branches are considered as optional, except that the product must be an instance of at least, one of them (and only one in the *only_one* case). These relationships can be formally specified as follows:

$$\_ \prec_R \_ : Product \leftrightarrow Relationship$$

$$\forall\, p : Product;\ t_i : FeatureTree;$$
$$t : \mathbb{F}_2\, FeatureTree \bullet$$
$$p \prec_R one\_or\_more(\ t\ ) \ \Leftrightarrow\ (\ \forall\, t_j : t \bullet\ p \prec_R optional(\ t_j\ )\ \wedge\ \exists\, t_k : t \bullet\ p \prec_\tau t_k\ )\ \wedge$$
$$p \prec_R only\_one(\ m\ ) \ \Leftrightarrow\ (\ \forall\, t_j : t \bullet\ p \prec_R optional(\ t_j\ )\ \wedge\ \exists_1\, t_k : t \bullet\ p \prec_\tau t_k\ )$$

For example, in Figure 9(a) a product is an instance of the *one_or_more* relationship r_4 if it is an instance of the D or E subtrees (or both). On the other hand, a product is an instance of the *only_one* relationship r_3 if it is an instance of only one of the F or G subtrees.

Finally, the $\prec$ predicate corresponding to CTCs (denoted as $\prec_\chi$), with the usual semantics of logical implication and mutual exclusion, is the following:

$$\_ \prec_\chi \_ : Product \leftrightarrow CTC$$

$$\forall\, p : Product;\ f_1, f_2 : Feature \bullet$$
$$p \prec_\chi requires(\ f_1, f_2\ ) \ \Leftrightarrow\ (\ f_1 \notin p\ \vee\ f_2 \in p\ )$$
$$\wedge$$
$$p \prec_\chi excludes(\ f_1, f_2\ ) \ \Leftrightarrow\ (\ f_1 \notin p\ \vee\ f_2 \notin p\ )$$

Once the abstract type *Model* and the operations Φ (*features–in–a–model*) and $\prec$ (*is–instance–of*) have been made concrete for BFM in the CML, all the analysis operations defined in the AFL (see Section 3) can be used without modification. Apart from this high level of reuse, other BFM–specific operations like those commented in [9] can also be defined in the CML. For the specification of other VMLs in the CML, see Appendices A and B.

5 Test–based validation of the FLAME framework

In order to validate the FLAME framework, the Prolog animation was tested using a fully automated approach following the IX commandment proposed in [13], which stresses the importance of testing when using formal methods. The goal of the test–based validation was twofold: (1) detecting flaws in the Z specification, i.e. mismatches between what was intended to be specified —the requirements describing in natural language the *intuitive* semantics in [9]— and the actual specification —the *formal* semantics described in Z in this article; and (2) detecting mismatches between the formal specification and its animation. In the following sections, the testing approach used, the validation setup, and the main results obtained during the validation of the FLAME framework are presented.

5.1 Metamorphic testing on the analysis of feature models

For the test–based validation of FLAME, the metamorphic test data generator presented by some of the authors in [64] was used. This generator, which is publicly available as a part of the open–source Java framework BeTTy [63], relies on a set of *metamorphic relations* between FMs and their corresponding set of valid products.

More specifically, a metamorphic relation is defined for each type of relationship among features, i.e. mandatory, optional, or, alternative, requires and excludes. Roughly speaking, these relations are based on the fact that when a variability constraint is added to an FM M, the set of valid products of the resulting *neighbor* model M' can be derived from the original one. As an example, consider the neighbor FMs and their associated product sets in Figure 10, where M' is derived from M by adding a mandatory feature D as a child of feature B. According to the semantics described in Section 2.1, the set of products of M' can be derived by adding the new mandatory feature D in all the products of M where its parent feature B appears.

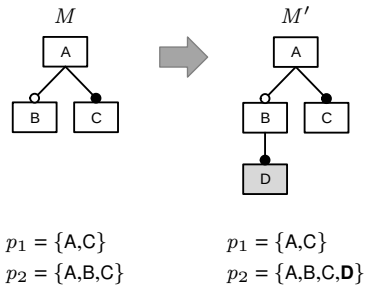


Fig. 10 Neighbor feature models after a mandatory feature (in gray) is added

Formally, let f_m be the mandatory feature added to M and f_p its parent feature. Then, the metamorphic relation between the products of M and those of M' can be defined as shown below (see [62] for the formal definition of all metamorphic relationships in FMs).

$$\#\Pi(M') = \#\Pi(M) \ \wedge$$
$$\forall\, p \in \Pi(M) \bullet f_p \notin p \Rightarrow p \in \Pi(M') \ \wedge$$
$$f_p \in p \Rightarrow (\, p \cup \{f_m\}\,) \in \Pi(M')$$

The metamorphic relations are used together with model transformations to generate FMs and their respective set of products, as shown in Figure 11. The process starts with an input FM whose set of valid products is known, i.e. a *seed*, which can be randomly generated from scratch, as in our approach, or obtained from an existing test case. A number of step–wise transformations are then applied, producing a neighbor model as well as its corresponding set of products according to the metamorphic relations. This process is repeated until some desired properties, *e.g.* number of features, are achieved. In Figure 11, M_0 is progressively extended in four steps until obtaining M_4 with 8 features $\{$A..H$\}$ and one CTC representing 6 different products.

Finally, once a FM with the desired properties is created, it is used as a non–trivial input for the analysis. Simultaneously, its set of products is automatically inspected to obtain the output of a number of analysis operations. Considering the M_i models and the sets of products generated in Figure 11, together with the analysis operations described in Section 3, the expected output of all of them can be obtained by simply answering questions such as:

- *Is M_1 void?* No, its set of products is not empty.
- *How many different products does M_2 represent?* 8 different products.
- *Is p = $\{$A,B,F$\}$ a valid product of M_3?* No, it is not included in its set of products.
- *Which are the core features of M_4?* Features $\{$A,C$\}$.
- *What is the commonality of feature* B *in M_4?* Feature B is included in 5 out of the 6 products of the set. Therefore, its commonality is $5/6 = 83.3\%$.
- *Does M_3 contain any dead feature?* Yes, feature G is dead since it is not included in any of the products represented by M_3.

The effectiveness of this metamorphic testing approach was evaluated using mutation testing [64]. In particular, hundreds of artificial faults (i.e. *mutants*) were introduced into several subject programs checking how many of them were detected by the test data generator. The percentage of detected faults, i.e. *mutation coverage*, ranged between 98.7% and 100%, which supports the effectiveness of the approach.

5.2 Test–based validation setup

The formal specification and its Prolog animation were developed in parallel, so it was possible to manually develop and run small tests as long as the analysis operations were specified and translated into Prolog. Although these tests were not developed systematically, they were quite useful for illustrating discussions among the authors and for detecting some problems related to numerical indicators (see Section 5.3.2). Once the reference implementation was finished, the systematic test–based validation was performed in a three–step process described below.

5.2.1 Test cases generation

For each analysis operation, 1,000 test cases were automatically generated using BeTTy. Each test case was composed of a random input FM and an expected output. In the case

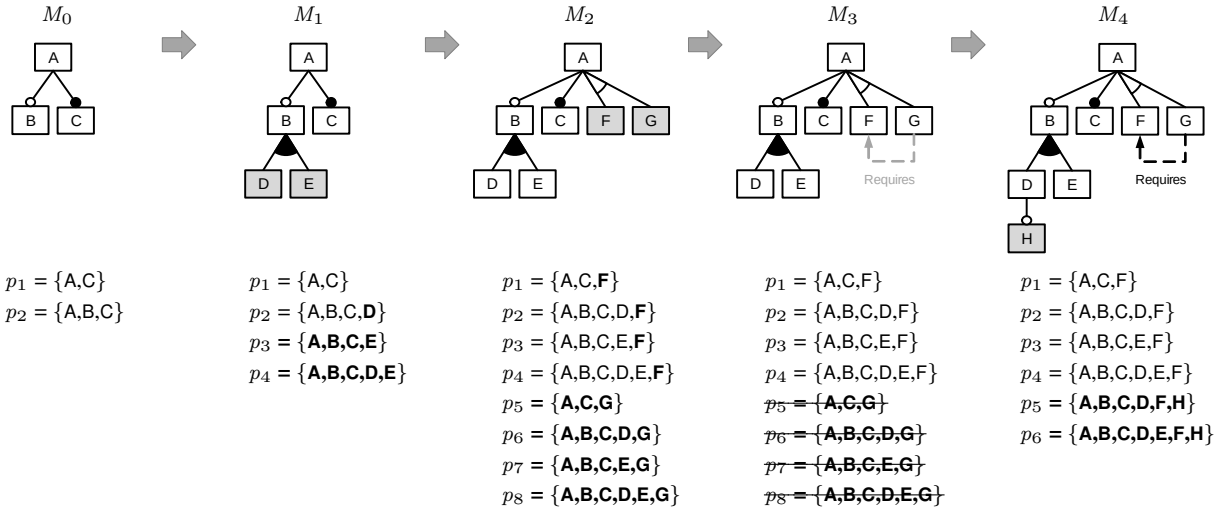


Fig. 11 Random generation of FMs and their set of products using metamorphic relations

of operations receiving other inputs apart from an FM (e.g. *valid product*, which takes a product to be checked), these inputs were generated using a *partition equivalence strategy* [6,46], e.g. generating valid and non–valid products with equal probability. For efficiency, FMs were generated with 10 features and 0–30% of CTCs with respect to the number of features. Previous works in testing by some of the authors of this article have shown that FMs with 10 features are complex enough to reveal faults effectively [64].

5.2.2 Tests execution in Prolog

Once the test cases for each operation were generated, they were translated into Prolog and integrated with the unit test framework described in [77]. An example of such integration for the *number of valid products* operation is shown in Figure 12. Once prepared, the tests were executed against the Prolog animation and the results were checked. Whenever a fault was detected, the Prolog animation and/or the formal specification were fixed and the tests were executed again. This process was repeated until obtaining a 100% of successful tests for all the analysis operations.

As reported by the Prolog unit test framework used, a 100% clauses coverage was achieved in the 20 Prolog files implementing the animation. Exceptionally, it was under 100% in two general–purpose files for managing sets and lists because only a portion of the code in those files is used by the animation. These results support the feasibility of the test–based validation approach and reinforce our confidence in the correctness of the reference implementation.

5.2.3 Tests execution in FaMa

Finally, all the generated test cases were executed against the FaMa framework [10,36], a mature framework previously developed by some of the authors based on a informal description of the analysis operations and well–known by the SPL community [71]. Using the Prolog animation as a reference implementation, the FaMa framework was checked in order detect possible deviations from expected results. Due to the maturity of the FaMa framework, this step was considered as a double–check for both frameworks.

In order to avoid biased results, the test–based validation process was performed by a group of authors completely independent from those in charge of the development of the formal specification and the reference implementation.

```
% SPL instances containing FMs generated by BeTTy
spl_db( spl_001, spl( ... ) ).
...
spl_db( spl_999, spl( ... ) ).

% Input data and expected result generated by BeTTy
test_data( spl_001, number_of_products, [], 12 ).
...
test_data( spl_999, number_of_products, [], 26 ).

% Test cases
:- begin_tests( number_of_products ).

% A test predicate for each test case that...
% 1. Retrieves an SPL instance
% 2. Retrieves input data and expected results
% 3. Performs the analysis operation
% 4. Compares the actual and expected results

test( number_of_products_001 ) :-
  spl_db( spl_001, SPL ),
  test_data( spl_001, number_of_products, [], EXPECTED ),
  nop( SPL, ACTUAL ),
  ACTUAL =:= EXPECTED.

test( number_of_products_002 ) :-
...

:- end_tests( number_of_products ).
```

Fig. 12 Structure of a Prolog unitary test for the validation of the *nop* (*number of products*) operation

5.3 Test–based validation results

The test–based validation of the FLAME framework revealed several flaws, especially in the previous informal definitions of some of the analysis operations in [9]. A description of these faults and how they were fixed are next presented.

5.3.1 Variant and dead features

In [9], *variant* features are defined as "those [features] that do not appear in all the products of an SPL". Taking this definition as a reference, the variant features (Φ_V) of an SPL were initially specified as all the features of an SPL except those that appear in all the products, i.e. all the features except the core features (Φ_C):

$$\Phi_V\ spl\ =\ spl.features \setminus \Phi_C\ spl$$

When the test–based validation was performed, it revealed that the previous definition considered dead features (i.e. those that do not appear in any product) as variant, even in the case of void SPLs, in which all their features are dead. After a discussion among the authors, the agreement on dead features not being variant was unanimous, so the informal definition in [9] was enhanced to explicitly declare that variant features cannot be dead and both the formal specification and its corresponding reference implementation were corrected (see Section 3.4.3 for the final definition and specification). This new definition of variant features implies that the core, variant and dead features are a *partition* of the feature set of an SPL, i.e. they are disjoint to each other and their union is the feature set:

$$\begin{aligned}&\forall\, spl : SPL \bullet \\ &\quad \Phi_C\ spl \cap \Phi_V\ spl\ =\ \varnothing\ \wedge \\ &\quad \Phi_C\ spl \cap \Phi_D\ spl\ =\ \varnothing\ \wedge \\ &\quad \Phi_V\ spl \cap \Phi_D\ spl\ =\ \varnothing\ \wedge \\ &\quad spl.features\ =\ \Phi_C\ spl \cup \Phi_V\ spl \cup \Phi_D\ spl\end{aligned}$$

Or more succinctly:

$$\forall\, spl : SPL \bullet \langle\, \Phi_C\ spl,\ \Phi_V\ spl,\ \Phi_D\ spl\, \rangle \text{ partitions } spl.features$$

This result has been incorporated into the AFL and reinforced by the addition of theorem 8 and its corresponding proof. See Section 3.4.3 and Appendix C for details.

5.3.2 Homogeneity and other numerical indicators

During the test–based validation, a problem with the definition of the *homogeneity* operation was detected. This operation was described in [9] as:

$$\mathcal{H}\ spl\ =\ 1\ -\ \frac{\#(\Phi_U\ spl)}{\mathcal{N}\ spl}$$

and the metamorphic tests made evident that if an SPL is void, its homogeneity cannot be computed because it includes a division by zero. After reviewing the original definition of the operation in [26], it was clear that there was a mistake in the definition of homogeneity in [9], so the formal specification was fixed in order to be compliant with their authors by substituting $\mathcal{N}\,spl$ by $\#spl.features$ in the denominator. However, as commented in Section 3.5.3, the definition of this indicator has been recently revised in [27], so its specification in the AFL has been updated accordingly.

The *homogeneity bug* was a signal to review all numerical indicators in which a division by zero was possible. This review led to enhanced definitions of *commonality* and *partial variability* (see Sections 3.5.1 and 3.5.2) with respect to the definitions in [9]. In the case of the former operation, it was not defined for void SPLs, whereas the latter was not defined for SPLs without variant features. In the FLAME framework, both operations are correctly specified, including those situations not previously considered. The FaMa framework [10,36] was also updated to be compliant with the new specification.

5.3.3 Atomic sets semantics

One of the most interesting results of the test–based validation was the difference between the semantics of the original, FM–dependent atomic sets operation and the same VML–independent operation defined in FLAME.

The concept of atomic set, as defined in [79,60] and as implemented in the FaMa framework, is defined only over FMs and it is based, not on a formal specification, but on an algorithm that merges parent features with their mandatory children features without considering CTCs. This concept of atomic sets allows an FM to be reduced by replacing groups of features by the corresponding atomic sets in its feature tree, as shown in Figure 13, borrowed from [9].

In the FLAME framework, the concept of atomic sets is specified not structurally but semantically in a higher–level, VML–independent manner (see Section 3.4.5), so it is ap-

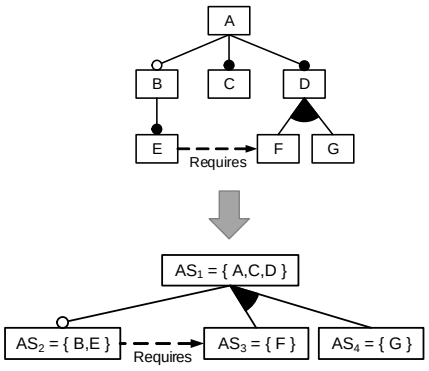


Fig. 13 Feature model reduction applying atomic sets

plicable not only to FMs but to any VML used as a characteristic model of an SPL. In FLAME, the atomic sets of an SPL are those maximal groups of features with a *subset or disjoint* semantics with respect to the SPL products: for every product, all the features in an atomic set appear together in the product, i.e. they are a *subset* of the product, or none of them appears at all, i.e. they are *disjoint* with the product.

With these semantics, both frameworks produced the same atomic sets during the test–based validation except when CTCs were relevant. For example, for the FM in Figure 13, the atomic sets produced by both frameworks are the same. If a new CTC (F requires E) is introduced, FaMa produces the same set of atomic sets, whereas FLAME produces {A,C,D}, {B,E,F} and {G} (see Figure 14).

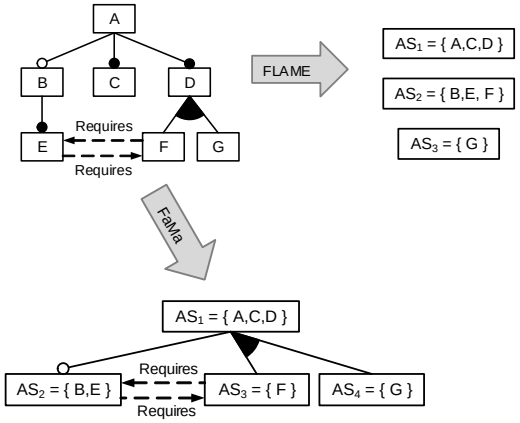


Fig. 14 Different atomic sets from the FaMa and FLAME frameworks

Notice that whereas the FaMa framework is able to reduce an FM using its atomic sets, the FLAME framework only computes the atomic sets, since is VML–independent. Notice also that those atomic sets computed by FLAME can be more accurate if CTCs are relevant. In general, both computations of atomic sets are interesting for different motivations, so the FaMa framework is being updated to include the new semantics but preserving the original ones.

5.3.4 Prolog toolkit for sets

In the early stages of the test–based validation, a fault in the Prolog toolkit developed for set theory was detected. The problem was related to the comparison of sets of sets (i.e. sets of *products*), that only worked if the sets to be compared were sorted in the same order. This fault was rapidly fixed so the Prolog animation could be systematically tested.

6 Related work

In a literature review developed by some of the authors [9], the formalization of analysis operations on FMs was identified as the main challenge in the field. In that work, up to 30 analysis operations were identified and informally reported and explained. In contrast to that work, the formal semantics of 20 out of the former 30 analysis operations, all those that could be reformulated in FM–independent manner, are provided in this article.

There are some proposals in the literature that have already defined formally, or at least with certain level of rigor, different analysis operations on FMs. They are summarized in Table 6 including the FLAME framework in the last row.

The first column in Table 6 (ABS) indicates if the proposal listed in the column is *abstract*, i.e. whether it specifies the semantics of the analysis operations without being coupled with any specific VML. In that sense, the FLAME framework, with all its analysis operations defined in the AFL (see Section 3), is a pioneer. Only Schobbens *et al.* [59] propose a sort of level of abstraction. In that work, a new FM notation called VFD (*Variant Feature Diagrams*) is defined and compared with other existing FM dialects. Their conclusion is that all the analyzed dialects can be translated into VFD, which is proved to be expressively complete and, as an example, some analysis operations are defined using it. In contrast, FLAME defines, among others, all the operations defined in [59] but at a much more abstract level, decoupled from any specific FM notation. Furthermore, the semantics of VFD could be specified in the CML of the FLAME framework applying the systematic approach defined in Section 4.

Table 6 Summary of the proposals reporting formalization of analysis operations on feature models

	ABS	RI	TBV	NOP
Zhang *et al.* [79]	-	+	-	4
Benavides *et al.* [8]	-	+	-	7
Von der Massen *et al.* [72]	-	-	-	1
Sun *et al.* [66]	-	+	∼	5
Fan *et al.* [25]	-	+	-	1
Gheyi *et al.* [32]	-	+	-	4
Bachmeyer *et al.* [4]	-	-	-	1
Schobbens *et al.* [59]	∼	-	-	3
Gheyi *et al.* [33]	-	+	-	2
Mendonca *et al.* [44]	-	-	-	4
Trinidad *et al.* [70]	-	+	-	4
Zhang *et al.* [78]	-	+	-	3
Fernandez *et al.* [26]	-	-	-	3
White *et al.* [76]	-	+	-	1
FLAME	**+**	**+**	**+**	**20**

The second column in Table 6 (RI) indicates if the proposal has a reference implementation derived from its formalization. There are some proposals that include an implementation of their formalization, but in most of the cases it is because the formalization is based on the underlying paradigm of the implementation platform. For instance, in [8] a *constraint satisfaction problem* (CSP) solver is used to implement seven operations defined using CSP primitives. Likewise, Fan and Zhang [25] use Description Logic to specify some analysis operations and a description logic reasoner as the implementation platform. In contrast, FLAME uses Z as an independent specification language and its corresponding Prolog animation as a reference implementation. Only Sun *et al.* [66] follow a similar approach. In that work, Z is also used to specify the semantics of FMs and an implementation using Alloy [38] is provided. However, the Z specification developed in [66] is FM–dependent, it cannot be extended to formalize other VMLs in a systematic way like FLAME, and the number of specified analysis operations is also lesser than in FLAME.

The third column in Table 6 (TBV) indicates if the proposal has been validated using a test–based approach. With respect to this, FLAME is the only proposal that, to the best of our knowledge, has applied an automated, systematic, test–based validation to its formal specification. Again, only Sun *et al.* [66] follow a similar approach performing a double validation. On the one hand, they develop 40 theorems and prove them using Z/EVES [55], although many of the them are merely auxiliary theorems to make the automatic proof possible. On the other hand, they animate their Z specification in Alloy and use a sample FM to test it. In both cases, they do not follow a systematic and automated approach as the described in Section 5, so their validation process cannot be considered as thorough as the one performed for FLAME.

Finally, the fourth column in Table 6 (NOP) shows that FLAME has the highest number of analysis operations specified, all of them in a VML–independent manner. As commented at the end of Section 4, it is possible to specify FM–dependent analysis operations in the CML of FLAME like *false optional features*, *conditionally dead features*, and others described in [9], although in this article only operations in the AFL of FLAME have been included in order to avoid an excessive length.

There are other VMLs that support some analysis operations. For instance, TVL (*Text–based Variability Language*) is a textual VML inspired in FMs whose syntax and semantics are formally presented in [18]. The semantics of TVL are represented as the set of products described by an instance of the language. In contrast to FLAME, TVL's formal semantics is coupled with a specific language, the formal semantics of operations are not discussed, and it has not been validated by systematic testing. FAMILIAR [1] is another textual VML that supports analysis capabilities. Similar to TVL, FAMILIAR is coupled with a specific notation, no formal semantics of the analysis operations are provided, and no testing–based validation has been performed.

Apart from the previously mentioned, there is a new stream of works that are extending the catalog of 30 analysis operations on FMs initially described in [9], especially from the testing area, in which the problem of selecting a set of products to be tested from an FM can be seen as an analysis operation. Just to mention a few, there are works using multi–objective approaches [56,34,47,29,30] for test selection, while some others explore *t–wise* testing [24,49,50, 48]. Although out of the scope of this work, FLAME could also be used to formalize these new operations.

7 Conclusions and future work

7.1 Conclusions

The main challenge identified in [9] in their survey on 20 years of automated analysis of FMs —*formally describe all the operations of analysis [of FMs] and provide a formal framework for defining new operations*— has been successfully faced and has resulted in the following contributions.

7.1.1 A reusable, verified formal framework

The developed formal framework, FLAME, can be used to formally specify not only FMs, but other VMLs as well. This reusability is achieved by its two–layered architecture. The AFL is the bottom layer which includes not only the definitions of necessary abstract concepts that can or must be redefined in the second layer, but also 20 VML–independent, therefore reusable, analysis operations. On top on the AFL, a family of CMLs, one for each VML to be formally specified, can be developed systematically. In this article, BFM, a FODA–like VML, has been formalized, but the same systematic approach could have been applied to other VMLs such us OVM [51] or CUDF [69] (see Appendices A and B for details on their ongoing formalization). Recently, an extension to the BFM specified in FLAME including non–functional feature attributes has been presented in [2].

7.1.2 An exhaustively tested reference implementation

In order to support tool development, a reference implementation, i.e. the result of the animation of the Z specification in Prolog, has also been developed as part of the FLAME framework. This reference implementation, currently used by students and tool developers, has been exhaustively tested, not only manually, but also automatically with more than 20,000 metamorphic test cases randomly generated using the BeTTy framework.

The AFL has also been formally verified by proving the 18 theorems that are included in Section 3 and proved in Appendix C.

7.1.3 Enhancements in SPL theory and tools

As a result of the successful integration of formal methods, i.e. Z and manual theorem proving on one hand, and automated metamorphic testing on the other hand, the following enhancements in SPL theory and tools have been achieved:

- New consistent semantics for the *core*, *dead* and *variant* features operations have been developed and incorporated into FaMa (see Section 5.3.1).
- The error in the description of the *homogeneity* operation in [9] and in its corresponding FaMa implementation has been fixed; its definition has been updated (see Section 5.3.2).
- New enhanced definitions of *commonality* and *partial variability* operations avoiding potential divisions by zero have been developed and their corresponding FaMa implementations upgraded (see Section 5.3.2).
- New VML–independent semantics for the *atomic sets* operation, which could lead to stronger model reductions in the future, have been developed and incorporated into FaMa while keeping also the FM–dependent version. As commented in section 5.3.3, both operation versions produce interesting results for SPL engineers.

7.2 Future work

The following lines of future work have been identified during the development of the FLAME framework.

7.2.1 Specification of more variability modeling languages

The success in the development of the FLAME framework invites to apply the same approach to the formalization of other VMLs. The formalizations of OVM [51] and CUDF [69] are advanced (see Appendices A and B for details). We are currently working on the identification of the metamorphic relationships in both notations in order to be able to automatically generate test cases with BeTTy [62,63].

We are also considering the formalization of other FM–like notations, such as those including cardinalities [22], or those extending features with numerical information in the form of attributes and relationships among them [8,54,2].

There is another family of VMLs known as *decision models* [58,57,21] that could be formalized using FLAME. Formalizing concepts for decisions models would be possible if we follow the approach described in [23], which allows transforming FMs to decision models and backwards, leading to the development of a CML for decision models. Textual VMLs have also been explored in the literature, such as FAMILIAR [1] and TVL [18]. Extending FLAME to support the semantics of those languages could be explored in the future.

7.2.2 Formalization of other analysis operations

Another interesting extension of FLAME is the formalization of VML–dependent analysis operations, some of them already mentioned in [9], such as *false optional features* or *conditionally dead features*. Also, as explained in Section 6, there is a new stream of work dealing with the selection of products for testing purposes. We can envision a FLAME extension to formalize testing–oriented analysis operations.

7.2.3 Redefining Π using metamorphic relationships

Although the reference implementation in FLAME does not have performance as one of its main goals, it is possible to improve performance significatively by redefining the *valid products* function (Π) in a VML–dependent manner. Currently, we are working on a redefinition of Π for those notations in which metamorphic relationships can be defined (see [62] for some examples), for example the FODA–like BFM notation formalized in this article.

7.2.4 Develop a more efficient atomic set algorithm

Another relevant point for performance improvement is the computation of the atomic sets of features. The abstract definition in Section 3.4.5 is elegant and easy to understand, but its algorithmic complexity is exponential because of the computation of the powerset of the features. We have designed a polynomial algorithm for the same computation which is currently under formalization and development.

7.2.5 Formalization of abstract features

Recently, the study of *abstract features*, i.e. features that appear in a variability model only to arrange other elements but with no associated semantics, has been recognized as an important challenge in [68] and [53]. The study of how to include abstract features in FLAME leads to another appealing future work.

7.2.6 Exploring other alternatives of animation

Exploring other alternatives for specification animation seems also interesting, being Alloy, Description Logic, and Maude the more likely candidates. The parallel development of the Z specification and its animation in Prolog generated a positive feedback between the two of them. Using other implementation platforms could bring new synergies and increase the quality of the FLAME framework in the future.

Acknowledgements The authors would like to thank José A. Galindo for his help implementing the BeTTy module for generating the tests in Prolog. We would also like to thank Miguel Toro, Pere Botella, Isidro Ramos, Frank van der Linden, Ernesto Pimentel, Vicente Pelechano, Daniel Le Berre, Sven Apel, Patrick Heymans, Paolo Borba, Maurice ter Beek, Rob Hierons, Michael Hinchey, and the anonymous reviewers, for their helpful comments on earlier versions of this article. Finally, we thank Marwa Benabdelali for using a very early version of the reference implementation of FLAME at the Institut Supérieur de Gestion de Tunis and provide early feedback.

References

1. M. Acher, P. Collet, P. Lahire, and R. France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78(6):657 – 681, 2013.
2. I. Achour, L. Labed, and H. Ben Ghezala. Towards an extended tool for analysis of extended feature models. In *Int. Symp. on Networks, Computers and Communications*, pages 1–5, June 2014.
3. AHEAD Tool Suite. `http://www.cs.utexas.edu/users/schwartz/ATS.html`, Accessed March 2015.
4. R. Bachmeyer and H. Delugach. A conceptual graph approach to feature modeling. In *Int. Conf. on Conceptual Structures*, pages 179–191, 2007.
5. D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conf.*, pages 7–20, 2005.
6. B. Beizer. *Software testing techniques*. Van Nostrand Reinhold Co., 2nd edition, 1990.
7. D. Benavides. *On the Automated Analysis of Software Product Lines using Feature Models*. PhD thesis, Univ. of Seville, 2007.
8. D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. In *Int. Conf. on Advanced Information Systems Engineering*, pages 491–503, 2005.
9. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 years Later: a Literature Review. *Information Systems*, 35(6):615–636, 2010.
10. D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a framework for the automated analysis of feature models. In *Int. Workshop on Variability Modeling of Software–Intensive Systems*, pages 129–134, 2007.
11. T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In *Int. Conf. on Automated Software Engineering*, pages 73–82, 2010.
12. BigLever. Biglever software gears. `http://www.biglever.com/`, Accessed March 2015.
13. J. Bowen and M. Hinchey. Ten commandments of formal methods... ten years on. In M. Hinchey and L. Coyle, editors, *Conquering Complexity*, pages 237–251. Springer–Verlag London, 2012.
14. W. Chan, S. Cheung, and K. Leung. A metamorphic testing approach for online testing of service-oriented software applications. *Int. Journal of Web Services Research*, 4(2):61–81, 2007.
15. T.Y. Chen, S.C. Cheung, and S.M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST–CS98–01, Univ. of Science and Tech., Hong Kong, 1998.
16. T.Y. Chen, J. Feng, and T.H. Tse. Metamorphic testing of programs on partial differential equations: a case study. In *Int. Computer Software and Applications Conf.*, pages 327–333, 2002.
17. T.Y. Chen, D.H. Huang, T.H. Tse, and Z.Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Ibero–American Symp. on Software Engineering and Knowledge Engineering*, pages 569–583, 2004.
18. A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming*, 76(12):1130–1143, 2011.
19. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison–Wesley, 2001.
20. W. F. Clocksin and C. S. Mellish. *Programming in Prolog: Using the ISO Standard*. Springer–Verlag, 5th edition, 2003.
21. K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski. Cool features and tough decisions: A comparison of variability modeling approaches. In *Int. Workshop on Variability Modeling of Software–Intensive Systems*, pages 173–182, 2012.
22. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
23. S. El-Sharkawy, S. Dederichs, and K. Schmid. From feature models to decision models and back again. In *Int. Software Product Line Conf.*, pages 126–135, 2012.
24. M. Fagereng Johansen, O. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. *Int. Software Product Line Conf.*, 2012.
25. S. Fan and N. Zhang. Feature model based on description logics. In *Int. Conf. on Knowledge–Based Intelligent Information and Engineering Systems*, pages 1144–1151, 2006.
26. D. Fernandez-Amorós, R. Heradio, and J. Cerrada. Inferring information from feature diagrams to product line economic models. In *Sofware Product Line Conf.*, pages 41–50, 2009.
27. D. Fernández-Amorós, R. Heradio, J.A Cerrada, and C. Cerrada. A Scalable Approach to Exact Model and Commonality Counting for Extended Feature Models. *IEEE Transactions on Software Engineering*, 40(9):895–910, Sept 2014.
28. Feature Modeling Plug–in. `http://gp.uwaterloo.ca/fmp/`, Accessed March 2015.
29. J. Galindo, M. Alférez, M. Acher, B. Baudry, and D. Benavides. A variability–based testing approach for synthesizing video sequences. In *Int. Symp. on Software Testing and Analysis*, pages 293–303, 2014.
30. J. Galindo, H. Turner, D. Benavides, and J. White. Testing variability intensive systems using automated analysis. an application in android. *Software Quality Journal*, Published online Nov. 2014.
31. J. García-Galán, P. Trinidad, O. F. Rana, and A. Ruiz-Cortés. Automated configuration support for infrastructure migration to the cloud. *Future Generation Computer Systems*, (In press), 2015.
32. R. Gheyi, T. Massoni, and P. Borba. A theory for feature models in alloy. In *First Alloy Workshop*, pages 71–80, 2006.
33. R. Gheyi, T. Massoni, and P. Borba. Algebraic laws for feature models. *J. of Univ. Computer Science*, 14(21):3573–3591, 2008.
34. C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. Le Traon. Multi–objective test generation for software product lines. In *Int. Software Product Line Conf.*, pages 62–71, 2013.
35. M. Hewitt, C. O'Halloran, and C. Sennett. Experiences with PiZA, an Animator for Z. In *Z User Meeting*, pages 35–51, 1997.
36. ISA Research Group. FaMa Tool Suite. Available at `http://www.isa.us.es/fama/`, Accessed March 2015.
37. ISO/IEC. Information technology — Z formal specification notation — Syntax, type system and semantics. International Standard ISO/IEC 13568:2002, 2002.
38. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edition, 2012.
39. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Soft. Engineering Institute, 1990.
40. P. King. Printing Z and Object–Z LaTeX documents. University of Queensland, 1990.
41. D. Le Berre and P. Rapicault. Dependency management for the eclipse ecosystem: eclipse p2, metadata and resolution. In *Int. Workshop on Open Component Ecosystems*, pages 21–30, 2009.
42. R. Lopez-Herrejon, L. Linsbauer, J. Galindo, J.A. Parejo, D. Benavides, S. Segura, and A. Egyed. An assessment of search–based techniques for reverse engineering feature models. *Journal of Systems and Software*, 103:353–369, 2015.

43. M. Mendonca, M. Branco, and D. Cowan. SPLOT: Software Product Lines Online Tools. In *Companion to the Int. Conf. on Object–Oriented Programming, Systems, Languages, and Applications*, pages 761–762, 2009.
44. M. Mendonca, A. Wasowski, K. Czarnecki, and D. Cowan. Efficient compilation techniques for large scale feature models. In *Generative Programming and Component Engineering Conf.*, pages 13–22, 2008.
45. C. Müller, M. Resinas, and A. Ruiz-Cortés. Automated Analysis of Conflicts in WS–Agreement. *IEEE Transactions on Services Computing*, 7(4):530–544, 2014.
46. G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.
47. R. Olaechea, S. Stewart, K. Czarnecki, and D. Rayside. Modeling and multi–objective optimization of quality attributes in variability–rich software. In *Int. Workshop on Non- functional System Properties in Domain Specific Modeling Languages*, 2012.
48. B. Pérez Lamancha and M. Polo Usaola. Testing product generation in software product lines using pairwise for features coverage. In *Int. Conf. on Testing Soft. and Systems*, pages 111–125, 2010.
49. G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 20(3–4):605–643, 2011.
50. G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. *Int. Conf. on Software Testing, Verification and Validation*, pages 459–468, 2010.
51. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer–Verlag, 2005.
52. pure::variants. `http://www.pure-systems.com/`, Accessed March 2015.
53. F. Roos-Frantz. *Automated Analysis of Software Product Lines with Orthogonal Variability Models*. PhD thesis, Univ. of Seville, 2012.
54. F. Roos-Frantz, D. Benavides, A. Ruiz-Cortés, A. Heuer, and K. Lauenroth. Quality-aware analysis in product line engineering with the orthogonal variability model. *Software Quality Journal*, 20(3–4):519–565, 2012.
55. M. Saaltink. The Z/EVES system. In *Z User Meeting*, pages 72–85. 1997.
56. A.S. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search–based software engineering: A case study in software product lines. In *Int. Conf. on Software Engineering*, pages 492–501, 2013.
57. K. Schmid and I. John. A customizable approach to full lifecycle variability management. *Science of Computer Programming*, 53(3):259–284, 2004.
58. K. Schmid, R. Rabiser, and P. Grünbacher. A comparison of decision modeling approaches in product lines. In *Work. on Variability Modeling of Software–Intensive Systems*, pages 119–126, 2011.
59. P. Schobbens, J.C. Trigaux P. Heymans, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.
60. S. Segura. Automated analysis of feature models using atomic sets. In *Workshop on Analyses of Software Product Lines*, pages 201–207, 2008.
61. S. Segura, D. Benavides, and A. Ruiz-Cortés. Functional Testing of Feature Model Analysis Tools: A Test Suite. *IET Software*, 5(1):70–82, 2011.
62. S. Segura, A. Durán, A.B. Sánchez, D. Le Berre, E. Lonca, and A. Ruiz-Cortés. Automated metamorphic testing of variability analysis tools. *Software Testing, Verification and Reliability*, 25(2):138–163, 2015.
63. S. Segura, J.A. Galindo, D. Benavides, J.A. Parejo, and A. Ruiz-Cortés. Betty: Benchmarking and testing on the automated analysis of feature models. In *Workshop on Variability Modelling of Software–intensive Systems*, pages 63–71, 2012.
64. S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated Metamorphic Testing on the Analyses of Feature Models. *Information and Software Technology*, 53(3):245–258, 2011.
65. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice–Hall, 2nd edition, 1992.
66. J. Sun, H. Zhang, Y.F. Li, and H. Wang. Formal semantics and verification for feature modeling. In *Int. Conf. on Engineering of Complex Computer Systems*, pages 303–312, 2005.
67. T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *Int. Conf. on Software Engineering*, pages 254–264, 2009.
68. T. Thüm, C. Kastner, S. Erdweg, and N. Siegmund. Abstract features in feature modeling. In *Software Product Lines Conf.*, pages 191–200, 2011.
69. R. Treinen and S. Zacchirol. Common upgradeability description format (cudf) 2.0. Technical Report 003, The Mancoosi project (FP7), 2009.
70. P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883–896, 2008.
71. P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A. Jimenez. Fama framework. In *Intl. Software Product Line Conf. – Tool Demonstrations*, page 359, 2008.
72. T. von der Massen and H. Litcher. Determining the variation degree of feature models. In *Software Product Lines Conf.*, pages 82–88, 2005.
73. M. M. West and B. M. Eaglestone. Software Development: Two Approaches to Animation of Z Specifications using Prolog. *Software Engineering Journal*, 7(4), 1992.
74. E.J. Weyuker. On testing non–testable programs. *The Computer Journal*, 25(4):465–470, 1982.
75. J. White, D. Benavides, D.C. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortés. Automated diagnosis of feature model configurations. *J. of Systems and Software*, 83(7):1094–1107, 2010.
76. J. White, J. Galindo, T. Saxena, B. Doughtery, D. Benavides, and D. Schmidt. Evolving feature model configurations in software product lines. *J. of Systems and Software*, 87:119–136, 2014.
77. J. Wielemaker. Prolog unit tests. Available at `http://www.swi-prolog.org/pldoc/package/plunit.html`, Accessed March 2015.
78. W. Zhang, H. Yan, H. Zhao, and Z. Jin. A BDD–based approach to verifying clone–enabled feature models' constraints and customization. In *Int. Conf. on Software Reuse*, pages 186–199, 2008.
79. W. Zhang, H. Zhao, and H. Mei. A Propositional Logic–Based Method for Verification of Feature Models. In *Int. Conf. on Formal Methods and Software Engineering*, pages 115–130, 2004.
80. Z.Q. Zhou, DH. Huang, TH. Tse, Z. Yang, H. Huang, and TY. Chen. Metamorphic testing and its applications. In *Int. Symp. on Future Software Technology*, pages 346–351, 2004.

Author Biographies

Amador Durán is an Associate Professor at the University of Seville (Spain), in which he received his Ph.D. and M.Sc. in Computer Science. He is also a senior member of the Applied Software Engineering research group (ISA, `www.isa.us.es`). His current research interests are requirements engineering, business process management, and software product lines.

David Benavides is an Associate Professor at the University of Seville (Spain), in which he obtained his Ph.D. in 2007. His main areas of expertise are Software Product Lines and Feature Model Analysis. He is also interested in open source and social applications of software systems. He has visited different research centers and has co–authored papers with more than forty external authors along the world and has special connections with South America.

Pablo Trinidad is a senior lecturer at the University of Seville, in which he received his Ph.D. and M.Sc. in Computer Science. He worked for several companies and as a freelance engineer before joining the academia in 2004. He is a member of the Applied Software Engineering research group (ISA, `www.isa.us.es`), in which he researches on the automated analysis of variability and service models with a special focus on automated deployment and maintenance of dynamic products.

Sergio Segura is a a full–time lecturer at the University of Seville, in which he received a Ph.D. in Software Engineering in 2011. His research interests include software testing, software variability and search-based software engineering. He has co–authored some highly-cited papers as well as tools used by universities and companies in various countries. He also serves regularly as a reviewer for international journals and conferences.

Antonio Ruiz–Cortésis an accredited Professor and Head of the Applied Software Engineering reserach group (ISA, `www.isa.us.es`) at the University of Seville (Spain), in which he received his Ph.D. and M.Sc. in Computer Science. His research interests are in the areas of service oriented computing, software product lines, and business process management.

A CML preview for OVM

This appendix contains the metamodel and the corresponding abstract syntax for the *Orthogonal Variability Modeling* (OVM) notation [51]. The complete CML specification, including the Φ (*features–in–a–model*) function and the $\prec\!\!<$ (*is–instance–of*) predicate has not been included in order to avoid an excessive length of the article.

The main concepts in OVM models are *variation points*, *variants* and *constraints*. Their graphical representation is shown in Figure 15, borrowed from [54]. The corresponding metamodel is shown in figure 16. For a thorough description of the OVM notation, the interested reader can consult [51].

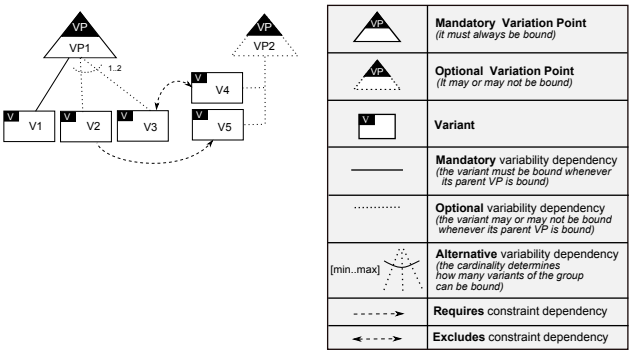


Fig. 15 OVM notation summary

The translation of the metamodel into an abstract syntax specification in Z is the following. First, an OVM model is defined as a nonempty set of *variation points* and a set of *constraints*.

$$Model \ ::= \ OVM \ \langle\!\langle \ \mathbb{F}_1 \, VariationPoint \times \mathbb{F} \, Constraint \ \rangle\!\rangle$$

Then, variation points are defined as mandatory and optional. In both cases, they are formed by a feature name and a nonempty set of *relationships*.

$$\begin{aligned} VariationPoint \ ::= \ & mandatory \langle\!\langle \ Feature \times \mathbb{F}_1 \ Relationship \ \rangle\!\rangle \\ | \ & optional \langle\!\langle \ Feature \times \mathbb{F}_1 \ Relationship \ \rangle\!\rangle \end{aligned}$$

The relationships between variation points and its *variants* are described as follows. Notice that the *alternative* relationship includes two natural numbers for the maximum and minimum cardinalities. Also, *variants* are described as containers of one feature name.

$$\begin{aligned} Relationship \ ::= \ & mandatory \ \langle\!\langle \ Variant \ \rangle\!\rangle \\ | \ & optional \ \langle\!\langle \ Variant \ \rangle\!\rangle \\ | \ & alternative \ \langle\!\langle \ \mathbb{F}_2 \ Variant \times \mathbb{N} \times \mathbb{N} \ \rangle\!\rangle \\ Variant \ ::= \ & variant \langle\!\langle \ Feature \ \rangle\!\rangle \end{aligned}$$

A generalization of variation points and variants, *variation element*, is needed to specify *constraints*, which are represented in a similar way as CTCs in BFM, except that in this case they can be set between any pair of variation elements, i.e. variation points and variants.

$$VariationElement \ ::= \ VariationPoint \ | \ Variant$$

$$\begin{aligned} Constraint \ ::= \ & \\ & requires \ \langle\!\langle \ VariationElement \times VariationElement \ \rangle\!\rangle \ | \\ & excludes \ \langle\!\langle \ VariationElement \times VariationElement \ \rangle\!\rangle \end{aligned}$$

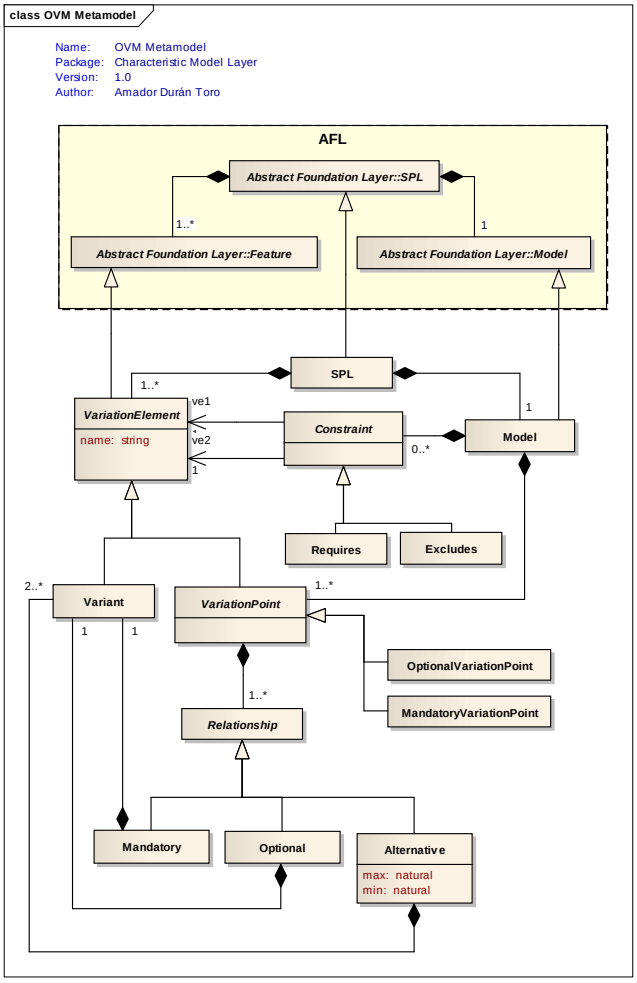


Fig. 16 OVM metamodel

B CML preview for CUDF

In a similar way to Appendix A, this appendix contains a preview of the CML for a simplified version of *Common Upgradeability Description Format* (CUDF) documents [69], a format for describing variability in package–based Free and Open Source Software (FOSS) distributions. A sample fragment of a CUDF document is shown in Figure 17.

Packages, attributed with name and version, are the main concept in CUDF documents, equivalent to *features* in BFM or OVM. They can be related to each other by *conflict* and *dependency* relationships. Dependency relationships can be grouped *conjunctively* —all dependencies must be satisfied— or *disjunctively* —at least one dependency must be satisfied. All relationships are version–dependant, both in depender and dependee packages. The corresponding metamodel is shown in figure 18.

```
...
package: arduino
version: 6
depends: libantlr-java > 4,
         openjdk-jdk | sun-java-jdk >= 6

package: php5-mysql
version: 5
depends: libc, libmysqlclient >= 5
conflicts: mysqli
...
```

Fig. 17 Sample CUDF document fragment

Before specifying the abstract syntax for CUDF documents, some preliminary definitions are needed. Assuming some type for package IDs (usually character strings), version numbers are defined as natural numbers, version comparators are defined as relations between pairs of version numbers, and features are redefined as $(PackageID, Version)$ pairs:

$$[PID] \quad \text{[Abstract type for package IDs]}$$
$$Version == \mathbb{N} \quad \text{[Versions are natural numbers]}$$
$$Comparator == Version \leftrightarrow Version \quad \text{[Version comparator]}$$
$$Feature ::= package\langle\langle PID \times Version \rangle\rangle \quad \text{[Features are packages in CUDF]}$$

Having defined previous concepts, a CUDF model can be defined as a set of package relationships:

$$Model ::= CUDF \langle\langle \mathbb{F}\ Relationship \rangle\rangle$$

Relationships, which can be *conflicts*, *conjunctive–dependencies*, or *disjunctive–dependencies*, are defined over *constraints* as follows:

$$\begin{aligned} Relationship ::= & \ conflict \langle\langle\ Constraint \rangle\rangle \\ | & \ conjunctiveDependency \langle\langle\ \mathbb{F}_1\ Constraint \rangle\rangle \\ | & \ disjunctiveDependency \langle\langle\ \mathbb{F}_1\ Constraint \rangle\rangle \end{aligned}$$

Finally, *constraints* are defined as 5–tuples (p, v, q, k, θ), where p and q are the identifiers of the depender and dependee packages respectively, v and k are literal version values, and θ is a comparison operator.

$$\begin{aligned} Constraint ::= & \ constraint \langle\langle \\ & PID \times Version \times PID \times Version \times Comparator \rangle\rangle \end{aligned}$$

For example, a constraint such as $(arduino, 2, JDK, 6, \geq)$ in a conjuctive dependency indicates that version 2 of the arduino package depends on the JDK package version 6 or higher.

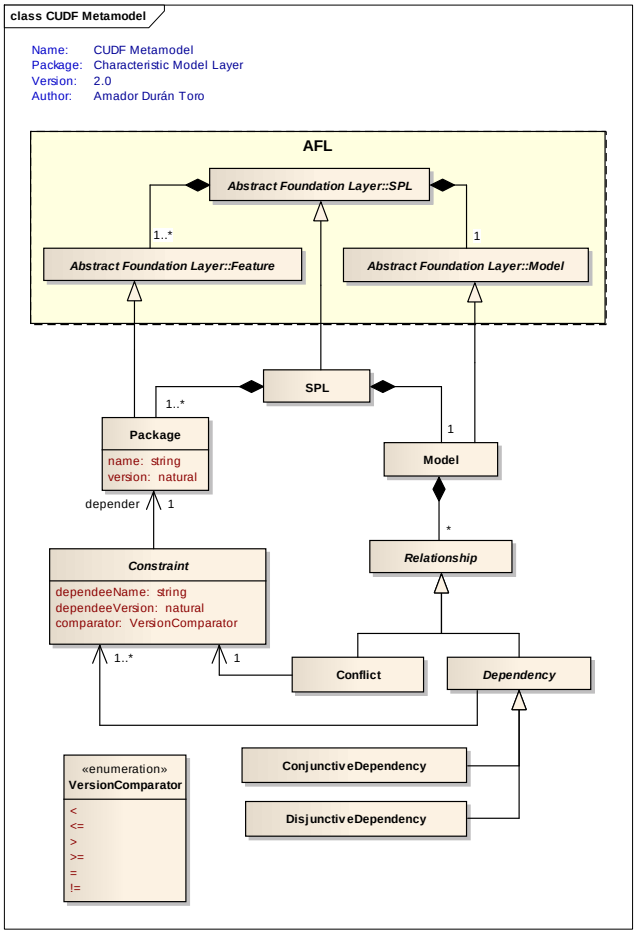


Fig. 18 CUDF metamodel

C Theorem proofs

This appendix contains the proof of theorems included in Section 3.

Proof of theorem 1 (the number of products of void SPLs is 0)

This theorem is proved by the substitution of $void$ and $\mathcal{N}$ by their definitions:

$$\forall\, spl : SPL \bullet void\ spl \Leftrightarrow \mathcal{N}\ spl = 0 \quad \text{[Theorem 1]}$$

$$\forall\, spl : SPL \bullet (\ \prod spl\ =\ \varnothing\ ) \Leftrightarrow (\ \#\prod spl\ =\ 0\ ) \quad [void \text{ \& } \mathcal{N} \text{ definitions}]$$

$$\forall\, spl : SPL \bullet (\ \#\prod spl\ =\ \#\varnothing\ ) \Leftrightarrow (\ \#\prod spl\ =\ 0\ ) \quad \text{[Apply \# in first term]}$$

Since $\#\varnothing = 0$ by definition, the theorem is proved. □

Proof of theorem 2 (there not exists any valid configuration for a void SPL)

This theorem is proved by the substitution of $\prec_c$ by its definition:

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \nexists\, c : Configuration \bullet c \prec_c spl \quad \text{[Theorem 2]}$$

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \nexists\, c : Configuration \bullet (\ selected\ c \cup removed\ c\ ) \subseteq spl.features \wedge \exists\, p : \prod spl \bullet p \lhd c \quad [\prec_c \text{ definition}]$$

$$\text{[False if } void\ spl\text{]}$$

Since spl is void, by definition $\prod spl$ is empty and therefore no valid product with respect to any configuration exists. □

Proof of theorem 3 (any filtering on a void SPL results in an empty set of products)

This theorem is proved by the substitution of $\prod_\sigma$ by its definition:

$$\forall\, spl : SPL;\ c : Configuration \bullet void\ spl \Rightarrow \prod_\sigma(\ spl, c\ ) = \varnothing \quad \text{[Theorem 3]}$$

$$\forall\, spl : SPL;\ c : Configuration \bullet void\ spl \Rightarrow \{\ p : \prod spl \mid p \lhd c\ \} = \varnothing \quad [\prod_\sigma \text{ definition}]$$

Since spl is void, by definition $\prod spl$ is empty and therefore $\{\ p : \prod\ spl\ \mid\ p \lhd c\ \}$ is also empty for any c. □

Proof of theorem 4 (any pair of void SPLs are equivalent)

This theorem is proved by the substitution of $\equiv$ by its definition:

$$\forall\, spl_1, spl_2 : SPL \bullet (\ void\ spl_1\ \wedge void\ spl_2\ ) \Rightarrow spl_1 \equiv spl_2 \quad \text{[Theorem 4]}$$

$$\forall\, spl_1, spl_2 : SPL \bullet (\ void\ spl_1\ \wedge void\ spl_2\ ) \Rightarrow \prod spl_1 = \prod spl_2 \quad [\equiv \text{ definition}]$$

Since spl_1 and spl_2 are void, by definition $\prod spl_1$ and $\prod spl_2$ are empty and therefore equal. □

Proof of theorem 5 (the set of core features of a void SPL is empty)

This theorem is proved by the substitution of Φ_C by its definition:

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \Phi_C\ spl\ =\ \varnothing \quad \text{[Theorem 5]}$$

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \bigcap \prod spl\ =\ \varnothing \quad [\Phi_C \text{ definition}]$$

Since spl is void, by definition $\prod spl$ is empty and therefore $\bigcap \prod spl$ is also empty. □

Proof of theorem 6 (all features of a void SPL are dead)

This theorem is proved by the substitution of Φ_D by its definition:

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \Phi_D\ spl\ =\ spl.features \quad \text{[Theorem 6]}$$

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow (\ spl.features \setminus \bigcup \prod spl\ )\ =\ spl.features \quad [\Phi_D \text{ definition}]$$

Since spl is void, by definition $\prod spl$ is empty and therefore $\bigcup \prod spl$ is also empty. □

Proof of theorem 7 (the set of variant features of a void SPL is empty)

This theorem is proved by the substitution of Φ_V by its definition:

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \Phi_V\, spl\ =\ \varnothing \quad \text{[Theorem 7]}$$

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow (\ spl.features\ \setminus \Phi_C\, spl\ \setminus \Phi_D\, spl\ ) = \varnothing \quad [\Phi_V \text{ definition}]$$

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow (\ spl.features\ \setminus \varnothing\ \setminus spl.features\ ) = \varnothing \quad \text{[Th. 5 \& 6]}$$

We know by theorems 5 and 6 that the set of core features of a void SPL is empty and that all its features are dead. Substituting in the subtraction expression of the three sets the theorem is proved. □

Proof of theorem 8 (the core, variant, and dead features of an SPL partition its features)

In order to prove this theorem, first we substitute the partitions expression by its definition and then the four resulting lemmas are proved:

$$\forall\, spl : SPL \bullet \langle\ \Phi_C\ spl,\ \Phi_V\ spl,\ \Phi_D\ spl\ \rangle \text{ partitions } spl.features \quad \text{[Theorem 8]}$$

$$\forall\, spl : SPL \bullet \Phi_C\ spl \cap \Phi_V\ spl\ =\ \varnothing\ \wedge \quad [\text{partitions definition}]$$

$$\Phi_C\ spl \cap \Phi_D\ spl\ =\ \varnothing\ \wedge \quad \text{[Lemma 1]}$$

$$\Phi_V\ spl \cap \Phi_D\ spl\ =\ \varnothing\ \wedge \quad \text{[Lemma 2]}$$

$$spl.features\ =\ \Phi_C\ spl \cup \Phi_V\ spl \cup \Phi_D\ spl \quad \text{[Lemma 3]}$$

$$\text{[Lemma 4]}$$

Lemma 1 (core and variant features are disjoint) *This lemma is proved by the substitution of Φ_V by its definition:*

$$\forall\, spl : SPL \bullet \Phi_C\ spl \cap \Phi_V\ spl = \varnothing \qquad \textit{[Lemma 1]}$$

$$\forall\, spl : SPL \bullet \Phi_C\ spl \cap (\ spl.features \setminus \Phi_C spl \setminus \Phi_D spl\ ) = \varnothing \qquad \textit{[}\Phi_V \textit{ definition]}$$

$\Phi_C spl$ is subtracted from $spl.features$ in the right hand side of the intersection expression; therefore, the intersection is empty. □

Lemma 2 (core and dead features are disjoint) *This lemma is proved by the substitution of Φ_C and Φ_D by their definitions:*

$$\forall\, spl : SPL \bullet \Phi_C spl \cap \Phi_D spl = \varnothing \qquad \textit{[Lemma 2]}$$

$$\forall\, spl : SPL \bullet (\bigcap \Pi spl\ ) \cap (\ spl.features \setminus \bigcup \Pi spl\ ) = \varnothing \qquad \textit{[}\Phi_C, \Phi_D \textit{ defs]}$$

Since the distributed intersection of a set of sets is always a subset of the distributed union of the same set of sets, i.e. $(\bigcap \Pi spl\ ) \subseteq (\bigcup \Pi spl\ )$, the set difference $(spl.features \setminus \bigcup \Pi spl\ )$ does not contain any feature in $(\bigcap \Pi spl\ )$, and therefore, the result of the intersection is empty. □

Lemma 3 (variant and dead features are disjoint) *This lemma is proved by the substitution of Φ_V by its definition:*

$$\forall\, spl : SPL \bullet \Phi_V spl \cap \Phi_D spl = \varnothing \qquad \textit{[Lemma 3]}$$

$$\forall\, spl : SPL \bullet (\ spl.features \setminus \Phi_C spl \setminus \Phi_D spl\ ) \cap \Phi_D spl = \varnothing \qquad \textit{[}\Phi_V \textit{ definition]}$$

$\Phi_D spl$ is subtracted from $spl.features$ in the left hand side of the intersection expression; therefore, the intersection is empty. □

Lemma 4 (the core, variant and dead features are all the features) *This lemma is proved by the substitution of Φ_V by their definition:*

$$\forall\, spl : SPL \bullet spl.features = \Phi_C\ spl \cup \Phi_V\ spl \cup \Phi_D\ spl \qquad \textit{[Lemma 4]}$$

$$\forall\, spl : SPL \bullet spl.features = \Phi_C\ spl \cup (\ spl.features \setminus \Phi_C spl \setminus \Phi_D spl\ ) \cup \Phi_D\ spl \qquad \textit{[}\Phi_V \textit{ definition]}$$

Subtracting and adding the same set to another set leave the latter unmodified, i.e. $(X \setminus Y) \cup Y = X$. In the union expression, $\Phi_C spl$ and $\Phi_D spl$ are subtracted and added to $spl.features$, resulting in $spl.features$ and therefore making both sides of the equality expression the same. □

Once lemmas 1, 2, 3, and 4 are proved, theorem 8 gets proved too. □

Proof of theorem 9 (the set of unique features of a void SPL is empty)

This theorem is proved by the substitution of Φ_U by its definition:

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \Phi_U spl = \varnothing \qquad \textit{[Theorem 9]}$$

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \{\ f_u : spl.features \mid \exists_1\ p : \Pi spl \bullet f_u \in p\ \} = \varnothing \qquad \textit{[}\Phi_U \textit{ definition]}$$

Since spl is void, by definition $\Pi\, spl$ is empty and therefore $(\exists_1\ p : \Pi\ spl \bullet f_u \in p)$ is false, making $\Phi_U spl$ empty. □

Proof of theorems 10 & 11 (in SPLs with more than one product, unique features are variant features & in SPLs with only one product, unique features are core features)

These theorems are proved together using the definition of Φ_V:

$$\forall\, spl : SPL \bullet \mathcal{N}\ spl > 1 \Rightarrow \Phi_U\ spl \subseteq \Phi_V\ spl \qquad \textit{[Theorem 10]}$$

$$\forall\, spl : SPL \bullet \mathcal{N}\ spl = 1 \Rightarrow \Phi_U\ spl = \Phi_C\ spl \qquad \textit{[Theorem 11]}$$

Considering the definition of Φ_V as $(\ spl.features \setminus \Phi_C\ spl \setminus \Phi_D\ spl\ )$, the definition of set subtraction implies that variant features cannot be neither core nor dead features, i.e.:

$$\forall\, spl : SPL;\ f : \Phi_V spl \bullet f \notin \Phi_C\ spl \wedge f \notin \Phi_D\ spl$$

On the other hand, we know that unique features cannot be dead by definition, i.e.:

$$\forall\, spl : SPL;\ f : \Phi_U spl \bullet f \notin \Phi_D\ spl$$

Since we know by theorem 8 that core, variant and dead features form a partition over the set of features of an SPL, unique features must then be core or variant.

$$\forall\, spl : SPL;\ f : \Phi_U spl \bullet f \in \Phi_C spl \vee f \in \Phi_V\ spl$$

If a unique feature is core, that means that is present in all products. The only way of being present only in one product (unique) and in all products (core) at the same time is when there is only one product in the SPL.

$$\forall\, spl : SPL;\ f : \Phi_U spl \bullet \mathcal{N} spl = 1 \Leftrightarrow f \in \Phi_C spl \wedge f \notin \Phi_V\ spl$$

$$\forall\, spl : SPL;\ f : \Phi_U spl \bullet \mathcal{N} spl > 1 \Leftrightarrow f \notin \Phi_C spl \wedge f \in \Phi_V\ spl$$

By elimination, if there are more than one product in an SPL, unique features cannot be core and must therefore be variant. □

Proof of theorem 12 (the core features, if any, are always one of the atomic sets)

This theorem is proved by the substitution of Φ_A and Φ_A^0 by their definitions:

$$\forall\, spl : SPL \bullet \Phi_C\ spl \neq \varnothing \Rightarrow \Phi_C spl \in \Phi_A\ spl \qquad \textit{[Theorem 12]}$$

$$\forall\, spl : SPL \bullet \Phi_C\ spl \neq \varnothing \Rightarrow (\ \Phi_C spl \in \Phi_A^0 spl\ ) \wedge maximal(\ \Phi_C spl, \Phi_A^0 spl\ ) \qquad \textit{[}\Phi_A \textit{ definition]}$$

$$\forall\, spl : SPL \bullet \Phi_C\ spl \neq \varnothing \Rightarrow (\ \forall p : \Pi\ spl \bullet \Phi_C spl \subseteq p \vee \Phi_C spl \cap p = \varnothing\ ) \wedge maximal(\ \Phi_C spl, \Phi_A^0 spl\ ) \qquad \textit{[}\Phi_A^0 \textit{ definition]}$$

Since core features are included in all products, $(\ \Phi_C spl \subseteq p\ )$ is true for all products and therefore core features are potential atomic sets. On the other hand, they are maximal by definition ($\Phi_C spl = \bigcap \Pi spl$), i.e. if a bigger potential atomic set existed, the core features would not be the core features but a proper subset of themselves. □

Proof of theorem 13 (the dead features, if any, are always one of the atomic sets)

This theorem is proved by the substitution of Φ_A and Φ_A^0 by their definitions:

$$\forall\, spl : SPL \bullet \Phi_D\ spl \neq \varnothing \Rightarrow \quad \text{[Theorem 13]}$$
$$\Phi_D spl \in \Phi_A\ spl$$

$$\forall\, spl : SPL \bullet \Phi_D\ spl \neq \varnothing \Rightarrow \quad [\Phi_A \text{ definition}]$$
$$(\ \Phi_D spl \in \Phi_A^0 spl\ ) \wedge$$
$$maximal(\ \Phi_D spl, \Phi_A^0 spl\ )$$

$$\forall\, spl : SPL \bullet \Phi_D\ spl \neq \varnothing \Rightarrow \quad [\Phi_A^0 \text{ definition}]$$
$$(\ \forall p : \prod\ spl \bullet \Phi_D spl \subseteq p \ \vee\ \Phi_D spl \cap p = \varnothing\ ) \wedge$$
$$maximal(\ \Phi_D spl, \Phi_A^0 spl\ )$$

Since dead features are not included in any product, ($\Phi_D spl \cap p = \varnothing$) is true for all products and therefore dead features are potential atomic sets. On the other hand, they are maximal by definition ($\Phi_D spl = spl.features \setminus \bigcup \Pi\, spl$), i.e. if a bigger potential atomic set existed, the dead features would not be the dead features but a proper subset of themselves. □

Proof of theorem 14 (void SPLs only have one atomic set, its features)

This theorem is proved by applying the results of theorems 6 and 13:

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \quad \text{[Theorem 14]}$$
$$\Phi_A\ spl = \{\ spl.features\ \}$$

We know by theorems 6 and 13 that all the features of a void SPL are dead, and that dead features are always an atomic set:

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \quad \text{[Theorem 6]}$$
$$\Phi_D\ spl = spl.features$$

$$\forall\, spl : SPL \bullet \Phi_D\ spl \neq \varnothing \Rightarrow \quad \text{[Theorem 13]}$$
$$\Phi_D spl \in \Phi_A\ spl$$

Since $spl.features$ is not empty by definition, we can conclude that in void SPLs, $spl.features \in \Phi_A\ spl$. Obviously, if $spl.features$ is an atomic set, no other atomic sets can exist. □

Proof of theorem 15 (the total variability of a void SPL is 0)

This theorem is proved by the substitution of $\mathcal{V}$ by its definition:

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \quad \text{[Theorem 15]}$$
$$\mathcal{V}\ spl = 0$$

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \quad [\mathcal{V} \text{ definition}]$$
$$\frac{\mathcal{N}\ spl}{2^{\#spl.features} - 1} = 0$$

Since spl is void, we know by theorem 1 that $\mathcal{N}\ spl = 0$. Therefore, $\mathcal{V}\ spl = 0$. □

Proof of theorem 16 (the partial variability of a void SPL is 0)

This theorem is proved by applying the results of theorem 7:

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \mathcal{V}_\rho\ spl = 0 \quad \text{[Theorem 16]}$$

Since spl is void, we know by theorem 7 that $\Phi_V spl = 0$.

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \Phi_V spl = \varnothing \quad \text{[Theorem 7]}$$

Because of the definition of $\mathcal{V}_\rho$ (see Section 3.5.2), $\Phi_V spl = 0$ implies that $\mathcal{V}_\rho\ spl = 0$. □

Proof of theorem 17 (The old homogeneity of a void SPL is 100%)

This theorem is proved by the substitution of $\mathcal{H}_{old}$ by its definition:

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \quad \text{[Theorem 17]}$$
$$\mathcal{H}_{old}\ spl = 100\%$$

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \quad [\mathcal{H}_{old} \text{ definition}]$$
$$1 - \frac{\#(\Phi_U\ spl)}{\#spl.features} = 100\%$$

Since spl is void, we know by theorem 9 that $\Phi_U spl = \varnothing$. Therefore, $\mathcal{H}_{old}\ spl = 100\%$. □

Proof of theorem 18 (The new homogeneity of a void SPL is 0%)

This theorem is proved by the substitution of $\mathcal{H}$ by its definition:

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \quad \text{[Theorem 18]}$$
$$\mathcal{H}\ spl = 0\%$$

$$\forall\, spl : SPL \bullet void\ spl \Rightarrow \quad [\mathcal{H} \text{ definition}]$$
$$\frac{\sum_{f_i\ \in\ spl.features} \mathcal{C}(\ spl,\ (\{f_i\}, \varnothing)\ )}{\#spl.features} = 0\%$$

Because of the definition of $\mathcal{C}$ (see Section 3.5.1), the commonality of a void SPL is always 0. Therefore, $\mathcal{H} spl = 0\%$. □

D Prolog code of the reference implementation

This appendix contains the translation guidelines applied to the translation of the Z specification into Prolog, and an example of use of the Prolog reference implementation, which can be downloaded from `http://www.isa.us.es/flame`, together with the 20,000 metamorphic tests.

D.1 Z–to–Prolog translation guidelines

The main guidelines followed during the manual translation of the Z specification into Prolog are described below.

- Z sets are represented as Prolog lists without duplicates, something common in the animation of Z specifications in Prolog [35,73]. A small toolkit for those set operations not present in the SWI Prolog distribution was developed for that purpose.
- The *SPL* schema type was represented as the *functor* `spl(F,M)`, where `F` is the SPL feature set and `M` is the SPL characteristic model. Functors are the usual way of representing compound objects in Prolog, see [20] for details.
- The *Configuration* type is represented as the functor `configuration(S,R)`, where `S` is the set of selected features and `R` is the set of removed features.
- The $\prec\!\!\!\cdot$ relation (*is–instance–of*) is represented as the `instance_of(P,M)` predicate, where `P` is a product and `M` is a characteristic model.
- The Φ function is represented as the `features(M,F)` predicate, where `M` is a characteristic model and `F` is the set of features used in the model.
- As a general pattern, when some elements in a set must be selected by satisfying a predicate, i.e.:

  $$y_s = \{ x : x_s \mid P(x) \} \quad \text{[the set of all } x\text{'s in } x_s \text{ satisfying } P\text{]}$$

  this is translated into Prolog using the standard predicate `findall(X,G,L)` [20], which returns a list `L` with all the values of `X` that satisfy the, possibly compound, goal `G`. In this pattern, the goal is formed by the conjunction of the membership of `X` to `X_S` and the satisfaction of predicate `P` on `X`:

  ```
  findall( X, ( member( X, X_S ), P( X ) ), Y_S )
  ```

- Another pattern was applied for translating expressions using the universal quantifier over the elements of a set, i.e.:

  $$\forall x : x_s \bullet P(x) \qquad \text{[true if all } x\text{'s in } x_s \text{ satisfy } P\text{]}$$

  This is translated into Prolog using the common predicate `forall(C,P)`, which succeeds if all solutions of `C` satisfy predicate `P`. In this case, the condition is the membership of `X` to `X_S`, and the predicate is any predicate `P` on `X`:

  ```
  forall( member( X, X_S ), P( X ) )
  ```

D.2 Sample use of the FLAME framework

If an SPL designer would like to use FLAME to analyze her FMs, she should represent them in the Prolog format for the FLAME abstract syntax. For example:

```
% SPL instances: spl_db( ID, spl( Features, Model ) )

spl_db( survey_spl,
  spl(
    [
      mobile_phone, calls, gps, screen, basic,
      color, high_resolution, media, camera, mp3
    ],
    bfm( feature( mobile_phone,
      [
        mandatory( feature( calls, [] ) ),
        optional( feature( gps, [] ) ),
        mandatory( feature( screen,
          [ only_one( [
              feature( basic, [] ),
              feature( color, [] ),
              feature( high_resolution, [] )
            ] )
          ]
        ) ),
        optional( feature( media,
          [ one_or_more( [
              feature( camera, [] ),
              feature( mp3, [] )
            ] )
          ]
        ) )
      ] ),
      [
        excludes( gps, basic ),
        requires( camera, high_resolution ),
        requires( mp3, mp3 )
      ]
    )
  )
).
```

Then, she could use a predicate like this for analyzing her SPL:

```
% Sample usage from Prolog prompt: analyze( survey_spl ).

analyze( SPL_ID ) :-
  spl_db( SPL_ID, SPL ),

  write( 'Checking ' ), write( SPL_ID ), nl,
  check_spl( SPL ),

  write( 'Products of ' ), write( SPL_ID ), nl,
  products_verbose( SPL, PRDS ),

  nop( SPL, NOP ),
  write( 'Number of products = ' ), write( NOP ), nl,

  core_features( SPL, CORE ),
  write( 'Core features = ' ), write( CORE ), nl,

  variant_features( SPL, VARIANT ),
  write( 'Variant features = ' ), write( VARIANT ), nl,

  dead_features( SPL, DEAD ),
  write( 'Dead features = ' ), write( DEAD ), nl,

  unique_features( SPL, UNIQUE ),
  write( 'Unique features = ' ), write( UNIQUE ), nl,

  atomic_sets( SPL, ATOMS ),
  write( 'Atomic sets: '), write( ATOMS ), nl,

  variability( SPL, V ),
  write( 'Variability = ' ),
  format( '~2f%', V * 100 ), nl,

  partial_variability( SPL, PV ),
  write( 'Partial variability = ' ),
  format( '~2f%', PV * 100 ), nl,

  homogeneity_old( SPL, H_OLD ),
  write( '(old) Homogeneity = ' ),
  format( '~2f%', H_OLD * 100 ), nl,

  homogeneity( SPL, H ),
  write( '(new) Homogeneity = ' ),
  format( '~2f%', H * 100 ), nl,

  !, fail.
```

That would produce the output in Figure 19 after calling `analyze( survey_spl )` from the Prolog prompt.

```
?- analyze( survey_spl ).
Checking survey_spl
No (more) errors found in SPL.
Products of survey_spl
Computing products...
Checking 1023 potential products:
Product [mobile_phone,calls,screen,basic] is valid!
Product [mobile_phone,calls,screen,color] is valid!
Product [mobile_phone,calls,gps,screen,color] is valid!
Product [mobile_phone,calls,screen,high_resolution] is valid!
Product [mobile_phone,calls,gps,screen,high_resolution] is valid!
Product [mobile_phone,calls,screen,high_resolution,media,camera] is valid!
Product [mobile_phone,calls,gps,screen,high_resolution,media,camera] is valid!
Product [mobile_phone,calls,screen,basic,media,mp3] is valid!
Product [mobile_phone,calls,screen,color,media,mp3] is valid!
Product [mobile_phone,calls,gps,screen,color,media,mp3] is valid!
Product [mobile_phone,calls,screen,high_resolution,media,mp3] is valid!
Product [mobile_phone,calls,gps,screen,high_resolution,media,mp3] is valid!
Product [mobile_phone,calls,screen,high_resolution,media,camera,mp3] is valid!
Product [mobile_phone,calls,gps,screen,high_resolution,media,camera,mp3] is valid!
There are 14 products in the SPL.
Number of products = 14
Core features = [mobile_phone,calls,screen]
Variant features = [gps,basic,color,high_resolution,media,camera,mp3]
Dead features = []
Unique features = []
Atomic sets: [[gps],[mobile_phone,calls,screen],[basic],[color],[high_resolution],
[media],[camera],[mp3]]
Variability = 1.37%
Partial variability = 11.02%
(old) Homogeneity = 100.00%
(new) Homogeneity = 58.57%
false.
```

Fig. 19 Output of the execution of the analysis of a sample SPL in the FLAME reference implementation