FLAME: FAMA Formal Framework (v 1.0)

Amador Durán, David Benavides, Sergio Segura,

Pablo Trinidad and Antonio Ruiz-Cortés

{amador,benavides,sergiosegura,ptrinidad,aruiz}@us.es

Applied Software Engineering Research Group
University of Seville, Spain
March 2012

Technical Report ISA–12–TR–02

This report was prepared by the

Applied Software Engineering Research Group (ISA)
Department of computer languages and systems
Av/ Reina Mercedes S/N, 41012 Seville, Spain
http://www.isa.us.es/

Support: This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project SETI (TIN2009-07366) and by the Andalusian Government under projects ISABEL (TIC-2533) and THEOS (TIC-5906).

Contents

Abstract

Software product lines are rapidly gaining importance across different application domains. This software production paradigm focuses on the development of related software products using managed reusable assets instead of building each product from scratch. In the software product line community, feature models are recognized as one of the most used notation to represent variability in a product line and their automated analysis is a thriving research area. In a recent systematic literature review, some of the authors summarized the numerous contributions on the topic in the last 20 years and identified several challenges to be addressed. One of them was the lack of formal definitions of the analysis operations, for which most of the reviewed works only provided informal descriptions, leading to misunderstandings and implementation problems in tool development. To face this challenge, this article presents FLAME, a formal framework for the specification of analysis operations on feature models. Its main advantages lie in its formal semantics—described using the Z specification language—and in its high level of abstraction, which allows the reuse of the framework for the formalization of different feature model dialects or even for different variability notations. Furthermore, in order to assure the quality of the formal framework—and to provide a reference implementation for tool developers—, the Z specification has been animated in Prolog and automatically validated using 18,000 test cases automatically generated using metamorphic testing techniques. The results of the performed tests have helped not only to enhance the framework significantly but also to detect inconsistencies both in the previous informal definitions of the analysis operations and in current analysis tools, thus supporting the need for formal semantics.

Chapter 1

Introduction

Software product lines (SPLs) are used as a way of managing a set of distinctive software products in a concrete domain. Because of the ubiquity of software systems, the production of software is rapidly shifting to a mass–customization production paradigm where a common platform of features conforms the core of the product line and a set of product–specific features are tailored for specific domain needs. In outline, SPL engineering covers specific processes, methods, models, techniques and tools for supporting SPL adoption [Clements and Northrop, 2001; Pohl et al., 2005]. One of the important issues of this branch of software engineering is the availability of modeling facilities to express the inherent variability in a product line. To this end, Kang et al. [1990] proposed the so–called *feature models*, which are variability models that represent the set of products in an SPL in terms of their features and the relationships among them.

As an essential support for SPL engineers, the automated analysis of feature models is defined by Benavides et al. [2010] as the computer–aided extraction of information from feature models by means of *analysis operations*, such as determining the number of products represented by a feature model, detecting anomalies, or comparing two models and classifying their relationship. Manual computation of such analysis operations depends on model semantics and is error–prone, tedious, and even infeasible with large–scale feature models. Some of the authors recently performed a systematic literature review on 20 years of automated analysis of feature models where 30 different analysis operations were identified [Benavides et al., 2010]. The review also identified several important challenges that were not covered by existing research. One of them was the lack of formal or rigorous description of analysis operations, which has sometimes led researchers and tool developers to misunderstandings.

This article presents FLAME (*Fama formaL frAMEwork*), a formal framework for defining the semantics of feature model analysis operations using the standard, well–known Z formal specification language [Spivey, 1992; ISO, 2002]. FLAME is designed in two layers. The first one is the *abstract foundation layer*, which includes not only the definitions of necessary abstract concepts that can or must be redefined in the second layer, but also 18 notation–independent analysis operations. The second layer is the *characteristic model layer*, where the semantics of specific variability modeling notations are specified. In this article, the second layer is used to specify the semantics of an eclectic feature model dialect known as *basic feature model* [Benavides et al.,

2010], although it could have been used to specify any other feature model dialect or other variability modeling notations like OVM [Pohl et al., 2005].

FLAME encompasses not only a reusable formal specification of feature models analysis operations, but also a Prolog–based reference implementation. Traditionally, Prolog [Clocksin and Mellish, 2003] has been the choice for the animation of Z specifications [Hewitt et al., 1997; West and Eaglestone, 1992]. In the case of FLAME, the goal of the animation process was threefold. Firstly, to detect problems and inconsistencies in the Z specification by the manual execution of a small set of tests, which were very helpful during the discussions among the authors about the semantics of some operations. Secondly, to validate the specification by means of 18,000 test cases automatically generated using metamorphic testing techniques inspired by the previous work of some of the authors [Segura et al., 2011]. Thirdly, to provide a exhaustively–tested, 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 adopted mixed approach—formal and test–based at the same time—has promoted the detection, discussion and correction of a number of misconceptions among the informal definitions, the Z specification, the corresponding Prolog animation, and third–party analysis tools, thus increasing the final quality of all of the components in FLAME.

The reminder of the article is structured as follows: Section 2 provides the necessary background on feature models for those readers not familiar with the topic; Section 3 describes the first, notation–independent layer of the FLAME framework, the *abstract foundation layer*; Section 4 describes a concrete application of the second layer of FLAME to a specific feature model notation, namely the *basic feature model* notation; Section 5 describes the test–based validation process of the framework and their results; Section 6 comments the related work and finally, Section 7 presents the conclusions and the future work. The reference implementation of FLAME is also included as an electronic appendix and can be downloaded from `http://www.isa.us.es/flame`.

Chapter 2

Feature models background

As mentioned in the previous section, feature models are widely used in the SPL community 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 feature models are usually depicted is shown in Figure 2.1, where the feature model describes an SPL for mobile phones taken from [Benavides et al., 2010].

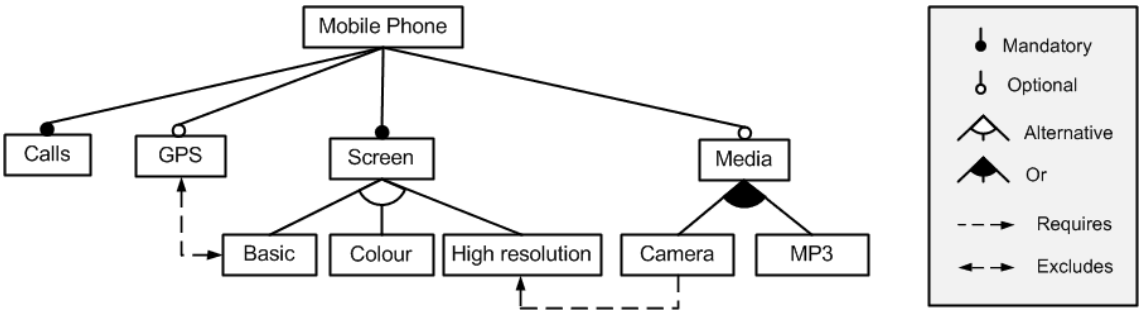

Figure 2.1: A sample feature model of an SPL for mobile phones

Although there are many proposals on the type of relationships and their graphical representation in feature models (see the work by Schobbens et al. [2007] for a detailed survey), the most usual relationships 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 2.1, `Calls` has a mandatory relationship with `Mobile phone` (its parent feature), 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 2.1, `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 2.1, `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`, `Colour` and `High resolution` have an alternative relationship in Figure 2.1, 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 2.1, `Camera` requires `High resolution` while `GPS` excludes `Basic`.

Chapter 3

Abstract foundation layer of the FLAME framework

The *abstract foundation layer* (AFL) is the basement where the FLAME framework relies on. In the AFL layer, some abstract concepts and most of the analysis operations described by Benavides et al. [2010] are formally defined. In the case of the analysis operations defined in the AFL, although they were originally defined over feature models, it has been possible to remove their dependencies from that specific SPL modeling notation and transform most of them into generic SPL analysis operations, i.e. they have been moved from the *characteristic model layer* (CML) into the AFL of FLAME.

As mentioned in Section 1, one of the objectives of the work presented in this article is not only the formal specification of the most used SPL analysis operations described by Benavides et al. [2010] *per se*, but also its automated validation by means of the application of the test suite described by Segura et al. [2011] to a *reference implementation* derived from the formal specification. For both purposes, the Z notation [Spivey, 1992; Woodcock and Davies, 1996] and the Prolog programming language [Clocksin and Mellish, 2003] have been chosen. On the one hand, the Z notation is one of the most used formal notations in software engineering and is based on first–order predicate logic and set theory, which makes the specification of SPL semantics quite straightforward. On the other hand, Prolog has a close relation to Z semantics, which makes it one of the most used programming languages for the so–called *specification animation* of Z specifications [Hewitt et al., 1997; West and Eaglestone, 1992].

In the rest of this section, all the concepts and operations related to SPL analysis from an abstract point of view are described threefold: (1) informally in natural language; (2) formally in Z; and (3) operationally in Prolog. For the formal specification, the name convention adopted is that operations related to *features* use the Greek letter *phi* (Φ), those related to *products* use the Greek letter *pi* ($\prod$), and operations yielding numbers use calligraphic letters such as $\mathcal{N}$. The Prolog code fragments are included in the descriptions in order to make the formal specification easier to understand for SPL tool developers and, in general, for readers not familiar with the Z notation.

3.1 SPL basic concepts

From a quite 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. 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 *models* must be redefined to describe the semantics of a concrete variability notation like feature models. A summary of the abstract concepts defined in the AFL, indicating whether they must be redefined in the CML or not, is shown in Table 3.1.

Name	Description	Redefinition in CML
SPL	Type for SPLs	optional
$Feature$	Type for features	optional
$Model$	Type for characteristic models	mandatory
$\_ \lessdot \_$	*is–instance–of* relation	mandatory
Φ	*features–in–a–model* function	mandatory

Table 3.1: Abstract concepts in the Abstract Foundation Layer of FLAME

3.1.1 SPL basic concepts in Z

To express the previously mentioned concepts in Z, the two abstract types $Feature$ and $Model$—i.e. *given sets* in Z terminology, see [Spivey, 1992] for details—are defined. Then, the $Product$ type is defined as a finite nonempty set of features.[1]

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

Another useful concept for SPL automated analysis is the so–called *configuration* [Benavides et al., 2010]. 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* during SPL analysis. This can be defined in Z as follows:

[1] 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 emptyset 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.

$$Configuration : \mathbb{F}\ Feature \times \mathbb{F}\ Feature \qquad \text{[Configuration type]}$$
$$selected, removed : Configuration \rightarrow \mathbb{F}\ Feature$$

$$\forall\, c : Configuration \bullet$$
$$selected\ c\ = first\ c \wedge$$
$$removed\ c\ = second\ c \wedge$$
$$selected\ c \cap removed\ c\ = \varnothing$$

where *first* and *second* are generic functions defined in [Spivey, 1992] to respectively access the first and second elements of any pair of objects.

Once the $Product$ and $Model$ types are defined, the abstract *is–an–instance–of* relation between products and models (denoted as $\prec\!\!<$, a symbol borrowed from the Object–Z notation [King, 1990]) can also be defined as follows:

$$\_ \prec\!\!< \_ : Product \leftrightarrow Model \qquad \text{[Abstract } \textit{is–an–instance–of} \text{ relation]}$$

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

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:

$$\Phi : Model \rightarrow \mathbb{F}\ Feature \qquad \text{[Abstract } \textit{features–in–a–model} \text{ function]}$$

[concrete definition must be provided in the CML]

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

SPL

$$model : Model \qquad \text{[SPL characteristic model]}$$
$$features : \mathbb{F}_1\ Feature \qquad \text{[SPL feature set]}$$

$$\Phi\ model = features \qquad \text{[model with known features only, no unbound features]}$$

[other constraints can be added in the CML]

which is considered as abstract in order to be augmented with additional constraints on features or models when used in the characteristic model layer of FLAME.

3.1.2 SPL basic concepts in Prolog

Since most of the basic concepts and operations defined in the previous section are abstract, almost no Prolog code can be associated to the Z specification yet. Nevertheless, some of the followed conventions are:

- Z sets are represented as Prolog lists, something common in the animation of Z specifications in Prolog [Hewitt et al., 1997; West and Eaglestone, 1992]. A small toolkit for set operations was developed for that purpose. The interested reader can consult the electronic appendix.
- The SPL schema type is represented as the *functor* `spl( F, M )`, where `F` is the SPL feature set and `M` is the SPL characteristic model. Functors are the way of representing compound objects in Prolog, see [Clocksin and Mellish, 2003] 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 $\prec\!\!\!<$ relation (*is–an–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.

3.2 SPL basic analysis operations

Once the main abstract concepts have been defined, a number of basic operations can be properly specified, following the naming conventions used by Benavides et al. [2010].

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 features in the SPL and is an instance of its characteristic model. As commented by Benavides et al. [2010], this operation may be helpful for SPL engineers to determine whether a given product is available in an SPL [White et al., 2010]. This can be expressed in Z as the following relation:

$$\_ \prec \_ : Product \leftrightarrow SPL \qquad \text{[Valid product]}$$

$$\forall\, p : Product;\ spl : SPL \bullet$$
$$p \prec spl \ \Leftrightarrow\ (\ p \subseteq spl.features \ \wedge\ p \prec\!\!\!< spl.model\ )$$

and transformed into the following Prolog predicate, which needs a *backtrackable*[2] version of the

[2] A *backtrackable* predicate can succeed more than once, thus providing all their solutions. See [Clocksin and Mellish, 2003] for details.

standard predicate `subset` in order to be used for computing the set of all valid products (see next subsection):

```
valid( P, spl( F, M ) ) :-
  btrck_subset( P, F ), % backtrackable version of subset
  instance_of( P, M ).  % abstract predicate
```

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 $\prod$), which may be helpful to identify new valid requirements combinations not considered in the initial scope of an SPL [Benavides et al., 2010], can be defined in Z as the following function:

$$\prod : SPL \rightarrow \mathbb{F}\, Product \qquad \text{[Valid products of an SPL]}$$

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

and in Prolog as follows, using the standard predicate `findall(X,P,L)` [Clocksin and Mellish, 2003], which returns a list `L` with all values for `X` that satisfy the predicate `P`.

```
products( spl( F, M ), PRDS ) :-
  findall( P, valid( P, spl( F, M ) ), PRDS ).
```

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 [Benavides et al., 2010], is the cardinality—denoted in Z as $\#S$—of its aforementioned set of products, i.e.:

$$\mathcal{N} : SPL \rightarrow \mathbb{N} \qquad \text{[Number of valid products]}$$

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

that can be easily transformed into the following Prolog predicate:

```
nop( spl( F, M ), NOP ) :-
  products( spl( F, M ), PRDS ),
  length( PRDS, NOP ).
```

3.2.4 Void SPL

An SPL is considered to be *void* if there not exists 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 [Benavides et al., 2010]. This can be expressed in Z by means of the following predicate:

$$void\_ : \mathbb{P}\, SPL \qquad \text{[Void SPL]}$$

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

and in Prolog as the following predicate:

```
void( spl( F, M ) ) :-
  products( spl( F, M ), [] ).
```

3.2.5 Full and partial configurations

As commented by Benavides et al. [2010], a product can always be transformed into a *full* configuration where the *selected* features are the product features and the *removed* features are the features not present in the product. More specifically, a configuration is said to be *full* or *partial* with respect to a given SPL if it uses all the features in the SPL or not, i.e.:

$$full, partial : Configuration \times SPL \qquad \text{[Full and partial relations]}$$

$$\forall\, c : Configuration;\ spl : SPL \bullet \\ \quad full(\ c, spl\ ) \Leftrightarrow (selected\ c \cup removed\ c) = spl.features \ \land \\ \quad partial(\ c, spl\ ) \Leftrightarrow (selected\ c \cup removed\ c) \subset spl.features$$

These relations can be transformed into the following Prolog predicates:

```
full( configuration( S, R ), spl( F, M ) ) :-
  union( S, R, U ),
  equal_set( U, F ).

partial( configuration( S, R ), spl( F, M ) ) :-
  union( S, R, U ),
  proper_subset( U, F ).
```

3.2.6 Validity of a configuration

As pointed out by Benavides et al. [2010], this operation is useful to provide the user with 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. Similarly to product validity for an SPL, 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 but none of the removed ones are. This can be specified in Z as follows:

$$\_ \lhd \_ : Product \leftrightarrow Configuration \qquad \text{[Valid product wrt a configuration]}$$

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

On the other hand, a configuration is considered as *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 in Z:

$$\_ \prec_c \_ : Configuration \leftrightarrow SPL \qquad \text{[Valid configuration wrt an SPL]}$$

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

and transformed into the following Prolog predicates:

```
valid_p_c( P, configuration( S, R ) ) :-
  subset( S, P ),
  intersection( R, P, [] ).

valid_c( configuration( S, R ), spl( F, M ) ) :-
  union( S, R, U ),
  subset( U, F ),
  products( spl( F, M ), PRDS ),
  member( P, PRDS ),
  valid_p_c( P, configuration( S, R ) ).
```

3.2.7 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 by Benavides et al. [2010], this operation may be helpful to assist users to select a desired product according to their key requirements. The specification of this operation in Z is as follows:

$$\prod_{\sigma} : SPL \times Configuration \rightarrow \mathbb{F}\ Product \qquad \text{[SPL product selection]}$$

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

Its corresponding Prolog predicate is the following, which requires the auxiliary predicate `filtered` in order to be used as the second argument of the `findall` predicate, in a similar way to the code in Section 3.2.2:

```
filter( spl( F, M ), configuration( S, R ), RESULT ) :-
  products( spl( F, M ), PRDS ),
  findall( P, filtered( P, configuration( S, R ), PRDS ), RESULT ).

filtered( P, configuration( S, R ), PRDS ) :-
  member( P, PRDS ),
  valid_p_c( P, configuration( S, R ) ).
```

3.3 SPL relations

As most software engineering products, SPLs usually experiment evolving changes during their development. In [Benavides et al., 2010], a number of relations between evolving SPLs are described that can help SPL engineers during the SPL development process.

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 the following relation:

$$\_ \equiv \_ : SPL \leftrightarrow SPL \qquad \text{[SPL equivalence]}$$

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

that can be directly translated into the following Prolog predicate:

```
equivalent( spl( F1, M1 ), spl( F2, M2 ) ) :-
  products( spl( F1, M1 ), PRDS1 ),
  products( spl( F2, M2 ), PRDS2 ),
  equal_set( PRDS1, PRDS2 ).
```

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:

$$
\begin{array}{|l}
\_ \sqsubset \_ : SPL \leftrightarrow SPL \qquad \text{[SPL specialization]} \\
\_ \sqsupset \_ : SPL \leftrightarrow SPL \qquad \text{[SPL generalization]} \\
\hline
\forall\, spl_1, spl_2 : SPL \bullet \\
\quad (spl_1 \sqsubset spl_2 \Leftrightarrow \prod spl_1 \subset \prod spl_2) \wedge \\
\quad (spl_2 \sqsupset spl_1 \Leftrightarrow spl_1 \sqsubset spl_2) \qquad \text{[inverse relations]}
\end{array}
$$

The corresponding Prolog predicates are the following:

```
specialization( spl( F1, M1 ), spl( F2, M2 ) ) :-
  products( spl( F1, M1 ), PRDS1 ),
  products( spl( F2, M2 ), PRDS2 ),
  proper_subset( PRDS1, PRDS2 ).

generalization( spl( F1, M1 ), spl( F2, M2 ) ) :-
  specialization( spl( F2, M2 ), spl( F1, M1 ) ).
```

3.3.3 SPL arbitrary edit

The last SPL evolving relation is the so–called *arbitrary edit*, which is the kind of relation between two SPL when they are neither equivalent nor a generalization or specialization of each other (see [Thüm et al., 2009] for details). This can be expressed in Z as the following relation:

$$
\begin{array}{|l}
\_\langle\langle\rangle\rangle\_ : SPL \leftrightarrow SPL \qquad \text{[SPL arbitrary edit]} \\
\hline
\forall\, spl_1, spl_2 : SPL \bullet \\
\quad spl_1 \langle\langle\rangle\rangle spl_2 \Leftrightarrow \\
\qquad \neg (spl_1 \equiv spl_2) \wedge \qquad \text{[not refactoring]} \\
\qquad \neg (spl_1 \sqsubset spl_2) \wedge \qquad \text{[not specialization]} \\
\qquad \neg (spl_1 \sqsupset spl_2) \qquad \text{[not generalization]}
\end{array}
$$

that can be directly translated into the following Prolog predicate:

```
arbitrary_edit( spl( F1, M1 ), spl( F2, M2 ) ) :-
  not( equivalent( spl( F1, M1 ), spl( F2, M2 ) ) ),
  not( specialization( spl( F1, M1 ), spl( F2, M2 ) ) ),
  not( generalization( spl( F1, M1 ), spl( F2, M2 ) ) ).
```

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.

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 by Benavides et al. [2010], 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 *distributed intersection*[3] (denoted as $\bigcap$) operator over the set of products:

$$\Phi_C : SPL \rightarrow \mathbb{F}\, Feature \qquad \text{[Core features]}$$

$$\forall\, spl : SPL \bullet$$
$$\Phi_C\ spl\ =\ \bigcap\ \prod\ spl$$

which is directly translated into the following Prolog predicate in which the version of the `intersection` predicate accepts also a set of sets (i.e. a list of lists) as its first argument:

```
core_features( spl( F, M ), C ) :-
  products( spl( F, M ), PRDS ),
  intersection( PRDS, C ).
```

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 *distributed union*[4] (denoted as $\bigcup$) over the set of products, i.e. all the features appearing in at least one valid product.

$$\Phi_D : SPL \rightarrow \mathbb{F}\, Feature \qquad \text{[Dead features]}$$

$$\forall\, spl : SPL \bullet$$
$$\Phi_D\ spl\ =\ spl.features \setminus \bigcup\ \prod\ spl$$

Again, its transformation into Prolog is straightforward:

[3]The distributed intersection (also known as *generalized intersection*) over A, being A a set of sets, is the set consisting of all objects belonging to every set in A.

[4]The distributed union (also known as *generalized union*) over A, being A a set of sets, is the set consisting of all objects belonging to any set in A.

```
dead_features( spl( F, M ), D ) :-
  products( spl( F, M ), PRDS ),
  union( PRDS, U ),
  subtract( F, U, D ).
```

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 not part of the *core* features and are not *dead* features either, something that can be easily expressed in Z by means of the set difference between the SPL features and its core and dead features:

$$\Phi_V : SPL \rightarrow \mathbb{F}\, Feature \qquad \text{[Variant features]}$$

$$\forall\, spl : SPL \bullet \Phi_V\ spl\ =\ spl.features \setminus \Phi_C\ spl \setminus \Phi_D\ spl$$

In a similar way to the two previous operations, the corresponding Prolog code is the following:

```
variant_features( spl( F, M ), V ) :-
  core_features( spl( F, M ), C ),
  dead_features( spl( F, M ), D ),
  subtract( F, C, VAUX ),
  subtract( VAUX, D, V ).
```

Notice that, as a result of the work described in this article, the definition of this operation differs from the presented in [Benavides et al., 2010]. See Section 5.3.1 for details.

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 quantifier*:

$$\Phi_U : SPL \rightarrow \mathbb{F}\, Feature \qquad \text{[Unique features]}$$

$$\forall\, spl : SPL \bullet \Phi_U\ spl\ =\ \{\, f : spl.features \mid \exists_1\, p : \prod spl \bullet f \in p \,\}$$

and transformed in the following Prolog code, in which two auxiliary predicates, `unique` and `contains`, have been introduced in order to use the standard `findall` predicate:

```
unique_features( spl( F, M ), RESULT ) :-
  products( spl( F, M ), PRDS ),
  findall( Fi, unique( Fi, F, PRDS ), RESULT ).

unique( Fi, F, PRDS ) :-
  member( Fi, F ),
  findall( P, contains( P, Fi, PRDS ), RESULT ),
  length( RESULT, 1 ).

contains( P, F, PRDS ) :-
  member( P, PRDS ),
  member( F, P ).
```

3.4.5 Atomic sets of features

First mentioned in [Zhang et al., 2004a] 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 [Benavides et al., 2010; Segura, 2008]. 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 [Benavides et al., 2010], 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 the 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, including all subsets with only one feature as its only member:

$$\Phi_A^0 : SPL \rightarrow \mathbb{F}\,\mathbb{F}_1\, Feature \qquad \text{[Potential atomic sets]}$$

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

The corresponding Prolog code is the following, in which two auxiliary predicates `potential_atomic_set` and `subset_or_disjoint` have been introduced to make possible the use of the standard predicates `findall` and `forall`:

```
potential_atomic_sets( spl( F, M ), POTATOMS ) :-
  products( spl( F, M ), PRDS ),
  findall( A0, potential_atomic_set( A0, F, PRDS ), POTATOMS ).
```

```
potential_atomic_set( A0, F, PRDS ) :-
  btrck_subset( A0, F ), % backtrackable version of subset
  A0 \= [],              % atomic sets are nonempty
  forall( member( P, PRDS ), subset_or_disjoint( A0, P ) ).

subset_or_disjoint( A, P ) :-
  subset( A, P )
  ;
  intersection( A, P, [] ).
```

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

$$[X]$$

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

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

and as the following Prolog predicate, where the existential quantifier has been transformed into the universal quantifier using the `forall` predicate and the logical equivalence $\nexists x \bullet P(x) \equiv \forall\, x \bullet \neg\, P(x)$:

```
maximal( Si, S ) :-
  member( Si, S ),
  forall( member( Sj, S ), not( proper_subset( Si, Sj ) ) ).
```

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

$$\Phi_A : SPL \rightarrow \mathbb{F}\, \mathbb{F}_1\, Feature \qquad \text{[Atomic sets]}$$

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

that can be easily transformed into the following Prolog code:

```
atomic_sets( spl( F, M ), ATOMS ) :-
  potential_atomic_sets( spl( F, M ), POTATOMS ),
  findall( A, maximal( A, POTATOMS ), ATOMS ).
```

Notice that the definition of this operation differs significatively from previous informal definitions provided by [Benavides et al., 2010], [Segura, 2008] and [Zhang et al., 2004b], which are feature–model 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.

3.5 SPL numerical indicators

Apart from the number of products ($\mathcal{N}$), defined in Section 3.2.3, other SPL numerical indicators which are not dependent on any modeling notation are defined in this section. As a result of the test–based validation of the FLAME framework, the definition of some indicators are enhanced with respect to the presented in [Benavides et al., 2010] in order correct some mistakes and to avoid division by zero in some quotients. See Section 5.3.2 for details.

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 [Benavides et al., 2010]. Its specification in Z is as follows:

$$\mathcal{C} : SPL \times Configuration \rightarrow \mathbb{R} \qquad \text{[commonality factor } \mathcal{C} \in 0..1\text{]}$$

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

and its operationalization in Prolog as follows:

```
commonality( spl( F, M ), configuration( _, _ ), 0 ) :-
  void( spl( F, M ) ).

commonality( spl( F, M ), configuration( S, R ), C ) :-
  filter( spl( F, M ), configuration( S, R ), FILTERED ),
  nop( spl( F, M ), NOP ),
  length( FILTERED, NOFP ),
  C is NOFP / NOP.
```

3.5.2 SPL variability

The *variability* of an SPL is defined in [Benavides et al., 2010] 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—one is subtracted from 2^n because the *empty product* is not considered as a valid product. 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 not variant features. Their Z specifications are the following:

$$\mathcal{V} : SPL \rightarrow \mathbb{R} \qquad \text{[Total variability } \mathcal{V} \in 0..1\text{]}$$

$$\forall\, spl : SPL \bullet$$
$$\mathcal{V}\ spl\ =\ \mathcal{N}\ spl\ /\ (\ 2^{\#spl.features} - 1\ )$$

$$\mathcal{V}_\rho : SPL \rightarrow \mathbb{R} \qquad \text{[Partial variability } \mathcal{V}_\rho \in 0..1\text{]}$$

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

and their corresponding transformations into Prolog are the following:

```
variability( spl( F, M ), V ) :-
  nop( spl( F, M ), N ),
  length( F, NOF ),
  V is N / ( 2**NOF - 1 ).

partial_variability( spl( F, M ), 0 ) :-
  variant_features( spl( F, M ), [] ).

partial_variability( spl( F, M ), PV ) :-
  nop( spl( F, M ), N ),
  variant_features( spl( F, M ), VF ),
  length( VF, NOVF ),
  PV is N / ( 2**NOVF - 1 ).
```

3.5.3 SPL homogeneity

According to Fernandez-Amorós et al. [2009] (but not to Benavides et al. [2010], 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 is 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:

$$\mathcal{H} : SPL \rightarrow \mathbb{R} \qquad [\mathcal{H} \in 0..1]$$

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

and directly transformed into Prolog as follows:

```
homogeneity( spl( F, M ), H ) :-
  unique_features( spl( F, M ), UF ),
  length( UF, NOUF ),
  length( F, NOF ),
  H is 1 - ( NOUF / NOF ).
```

Chapter 4

Characteristic model layer of the FLAME framework

The *characteristic model layer* (CML) is the layer of FLAME where the specifics aspects of different SPL modeling notations 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 $\prec\!\!\!\cdot$ (*is–an–instance–of*) have to be specified in order to formalize an SPL modeling notation.

Among the different SPL modeling notations, the *basic feature model* (BFM) as described by Benavides et al. [2010] has been chosen for its formalization in this article 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. Since the corresponding Prolog code is significantly longer than in the AFL operations, it has been omitted in this section, but the interested reader can consult the electronic appendix or download it from `http://www.isa.us.es/flame`.

4.1 BFM as a characteristic model

Following Benavides et al. [2010], 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* (CTC), 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. In Section 2, Figure 2.1 shows a sample BFM for an SPL for mobile phones.

4.1.1 BFM metamodel

The first task to formalize a modeling notation in FLAME is to specify the abstract type $Model$, which usually represents the metamodel of the modeling notation. For that purpose, the metamodel

in figure 4.1 was first developed in UML, then formalized in Z by means of the so–called *free types* [Spivey, 1992], which are usually used to specify *abstract syntaxes*, and finally translated into Prolog functors.

Following this approach, the main symbol in the abstract syntax of BFM models is used to redefine the abstract type $Model$ as a pair formed by a *tree feature*, i.e. the *root* feature, and a finite set of *cross–tree–constraints*:

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

Then, tree features are defined as *leaf features* ($feature_\lambda$) or as *compound features* ($feature_\kappa$), which in turn have a nonempty set of *mandatory*, *optional*, *one–or–more* or *only–one relationships*:

$$\begin{aligned} TreeFeature \ &::= \ feature_\kappa \langle\langle \ Feature \times \mathbb{F}_1 \ Relationship \ \rangle\rangle \\ &\mid \ feature_\lambda \langle\langle \ Feature \ \rangle\rangle \\ \\ Relationship \ &::= \ mandatory \ \langle\langle \ TreeFeature \ \rangle\rangle \\ &\mid \ optional \ \langle\langle \ TreeFeature \ \rangle\rangle \\ &\mid \ one\_or\_more \ \langle\langle \ \mathbb{F}_2 \ TreeFeature \ \rangle\rangle \\ &\mid \ only\_one \ \langle\langle \ \mathbb{F}_2 \ TreeFeature \ \rangle\rangle \end{aligned}$$

Notice that the *one–or–more* and *only–one* relationships are compound of a finite set of at least two tree features. 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 element apart from feature trees in BFMs are the cross–tree–constraints, which are simply defined as the CTC type formed by pairs of features which *require* or *exclude* each other:[1]

$$\begin{aligned} CTC \ &::= \ requires \ \langle\langle \ Feature \times Feature \ \rangle\rangle \\ &\mid \ excludes \ \langle\langle \ Feature \times Feature \ \rangle\rangle \end{aligned}$$

As an example of use, the BFM in figure 2.1 can be represented in the previously defined abstract syntax as follows:

[1] Other approaches propose the use of propositional logic, e.g. *well–formed–formulas*, for CTCs. See for example [Batory, 2005] or [Benavides, 2007], where a very preliminary version of this work includes them.

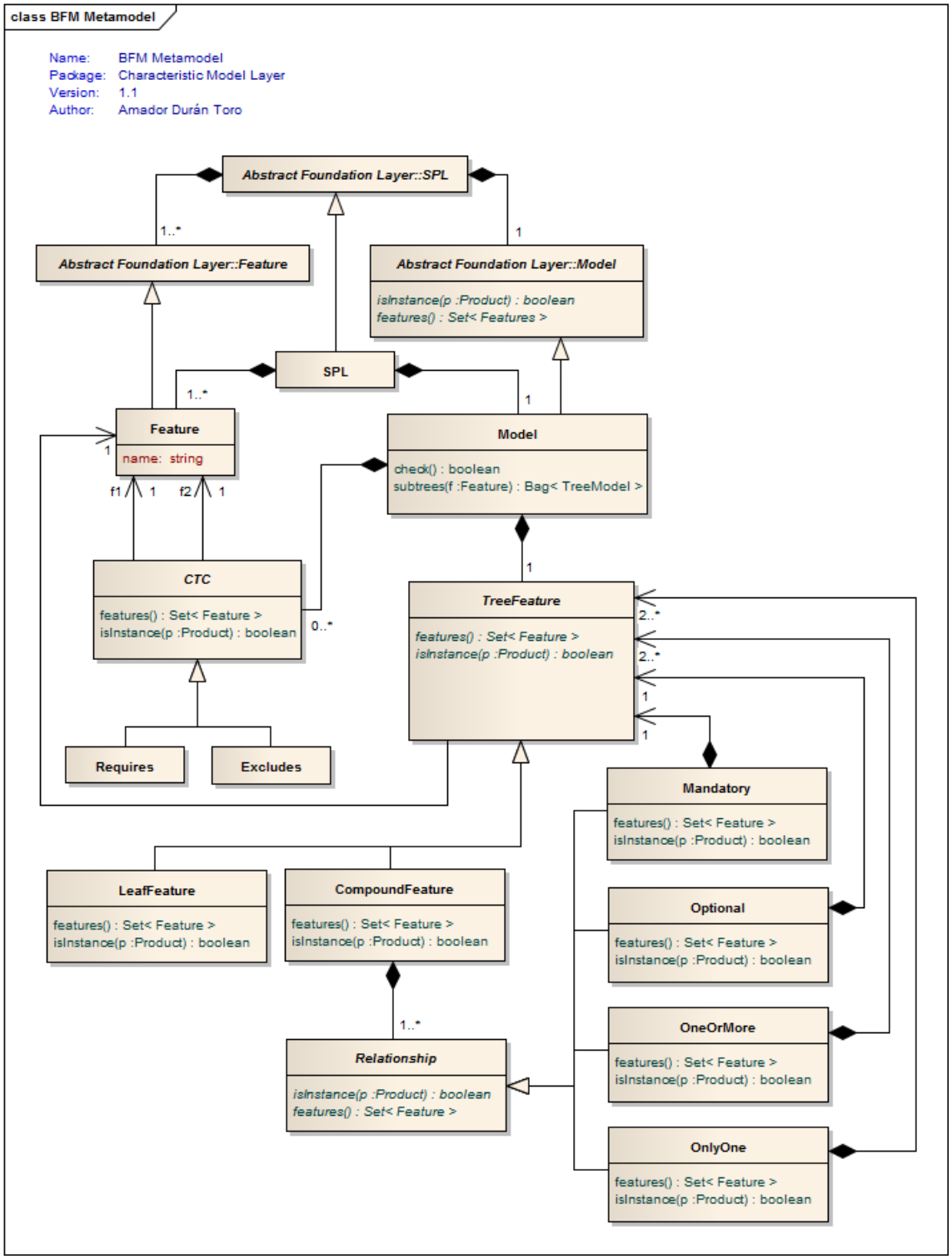

Figure 4.1: BFM metamodel

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

and as Prolog functors as follows:

```
bfm(
  k_feature( mobile_phone, [
    mandatory( l_feature( calls ) ),
    optional( l_feature( gps ) ),
    mandatory( k_feature( screen, [
      only_one( [ l_feature( basic ),
                  l_feature( colour ),
                  l_feature( high_resolution )
      ] )
    ] ) ),
    optional( k_feature( media, [
      one_or_more( [ l_feature( camera ),
                     l_feature( mp3 )
      ] )
    ] ) )
  ] ),
  [
    excludes( gps, basic ),
    requires( camera, high_resolution ),
  ]
)
```

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, i.e. the root feature, and the set of CTCs from a given BFM:

$$tree : Model \rightarrow TreeFeature \qquad \text{[Helper functions } \textit{tree} \text{ and } \textit{ctc}\text{]}$$
$$ctc : Model \rightarrow \mathbb{F}\, CTC$$
$$\forall\, t : TreeFeature;\ c : \mathbb{F}\, CTC;\ m : Model \bullet$$
$$tree\ BFM(t, c) = t \ \land$$
$$ctc\ BFM(t, c) = c$$

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

$$children : Relationship \rightarrow \mathbb{F}_1\ TreeFeature \qquad \text{[Helper function } \textit{children}\text{]}$$
$$\forall\, t_i : TreeFeature;\ t : \mathbb{F}_2\ TreeFeature \bullet$$
$$children\ mandatory(\ t_i\ ) = children\ optional(\ t_i\ ) = \{\ t_i\ \} \ \land$$
$$children\ one\_or\_more(\ t\ ) = children\ only\_one(\ t\ ) = t$$

The last helper functions are $\mathcal{F}_T$ and $\mathcal{F}_R$, which return the number of times a given feature appears in a $TreeFeature$ and in a $Relationship$. Its specification in Z[2] is as follows:

$$\mathcal{F}_\tau : Feature \times TreeFeature \rightarrow \mathbb{N}$$
[number of times a feature appears in a tree]
$$\mathcal{F}_R : Feature \times Relationship \rightarrow \mathbb{N}$$
[number of times a feature appears in a relationship]
$$\forall f_1,\ f_2 : Feature;\ r : \mathbb{F}\, Relationship \bullet$$
$$f_1 = f_2 \ \Rightarrow\ \mathcal{F}_\tau(\ f_1,\ feature_\lambda(\ f_2\ )\ ) = 1 \ \land$$
$$\mathcal{F}_\tau(\ f_1,\ feature_\kappa(\ f_2, r\ )\ ) = 1 + \sum_{r_i \in r} \mathcal{F}_R(\ f_1,\ r_i\ )$$
$$\forall f_1,\ f_2 : Feature;\ r : \mathbb{F}\, Relationship \bullet$$
$$f_1 \neq f_2 \ \Rightarrow\ \mathcal{F}_\tau(\ f_1,\ feature_\lambda(\ f_2\ )\ ) = 0 \ \land$$
$$\mathcal{F}_\tau(\ f_1,\ feature_\kappa(\ f_2, r\ )\ ) = \sum_{r_i \in r} \mathcal{F}_R(\ f_1,\ r_i\ )$$
$$\forall f : Feature;\ r : Relationship \bullet$$
$$\mathcal{F}_R(\ f,\ r\ ) = \sum_{t_i \in children\ r} \mathcal{F}_T(\ f,\ t_i\ )$$

[2]The use of the *summation symbol* (Σ) over the elements of a set is not explicitly defined in Z. Since its alternatives are complex, we have decided to use it for the sake of clarity.

4.2 Redefining the *SPL* type

As commented at the end of section 3.1.1, the abstract SPL schema type can be augmented with additional constraints related to the nature of features or models. In the case of BFM, its feature models are structurally trees, which implies that a feature cannot appear more than once in a model. Using the previously defined $\mathcal{F}_\tau$ function, this constraint can be added to the original SPL schema type resulting in the following:

SPL

$model : Model$ [SPL using BFM]

$features : \mathbb{F}_1\ Feature$

$\Phi\ model = features$

$\forall\ f : features \bullet$

$\quad \mathcal{F}_\tau(\ f,\ tree\ model\ )\ =\ 1$ [All features appear only once]

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

Having redefined the $Model$ and the SPL types for using BFM, the Φ function must also be redefined in order to make concrete the abstract definitions in the AFL. The first step is to specify that the features used in a BFM are the features used either in their tree feature model or in their CTCs, i.e.:

$\Phi : Model \rightarrow \mathbb{F}\ Feature$ [Redefinition of Φ for BFM]

$\forall\, m : Model \bullet$

$\quad \Phi\ m\ =\ \Phi_\tau\ tree\ m\ \cup\ \bigcup\{\ ctc_i : ctc\ m\ \bullet\ \Phi_\chi\ ctc_i\ \}$

The second step is to specify the Φ functions for determining the features in a BFM tree (denoted as Φ_τ) and in a $Relationship$ (denoted as Φ_R). The Z specification of both functions is the following:

$\Phi_\tau : TreeFeature \rightarrow \mathbb{F}\ Feature$ [Features in a BFM tree]

$\Phi_R : Relationship \rightarrow \mathbb{F}\ Feature$ [Features in a BFM relationship]

$\forall\ f : Feature;\ r_i : Relationship;\ r : \mathbb{F}\ Relationship \bullet$

$\quad \Phi_\tau\ feature_\lambda(\ f\ )\ =\ \{\ f\ \}\ \wedge$

$\quad \Phi_\tau\ feature_\kappa(\ f,\ r\ )\ =\ \{\ f\ \}\ \cup\ \bigcup\{\ r_i : r\ \bullet\ \Phi_R\ r_i\ \}\ \wedge$

$\quad \Phi_R\ r_i\ =\ \bigcup\{\ t_i : children\ r_i\ \bullet\ \Phi_\tau\ t_i\ \}$

Finally, the specification of the function returning the features in a CTC (denoted as Φ_χ) is the following:

$$\Phi_\chi : CTC \rightarrow \mathbb{F}\, Feature \qquad \text{[Features in a CTC]}$$

$$\forall\ f_1,\ f_2 : Feature \bullet$$
$$\Phi_\chi\ requires(\ f_1,\ f_2\ )\ =\ \Phi_\chi\ excludes(\ f_1,\ f_2\ )\ =\ \{\ f_1\ \}\ \cup\ \{\ f_2\ \}$$

4.4 Redefining the *is–instance–of* relation

The redefinition of the $\prec\!\!<$ relation is structurally very similar to the redefinition of the Φ function described in the previous section. 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.

$$\_ \prec\!\!< \_ : Product \leftrightarrow Model \qquad \text{[Redefinition of } \prec\!\!< \text{ for BFM]}$$

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

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 leaf feature if it includes the leaf feature, whereas is an instance of a compound feature if it includes the parent feature and is an instance of all its children relationships, i.e.:

$$\_ \prec\!\!<_\tau \_ : Product \leftrightarrow TreeFeature \qquad \text{[Instance of a BFM tree]}$$

$$\forall\, p : Product;\ f : Feature;\ r : \mathbb{F}\, Relationship \bullet$$
$$p \prec\!\!<_\tau feature_\lambda(\ f\ )\ \Leftrightarrow\ f \in p\ \wedge$$
$$p \prec\!\!<_\tau feature_\kappa(\ f,\ r\ )\ \Leftrightarrow\ (\ f \in p\ \wedge\ \forall\ r_i : r\ \bullet\ p \prec\!\!<_R r_i\ )$$

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 corresponding subtree; in the case of optional subtrees, a product is an instance if is an instance of the subtree or the set of features of the product and of the subtree are disjoint; 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).

$$\_ \prec\!\!<_R \_ : Product \leftrightarrow Relationship \qquad \text{[Instance of a BFM relationship]}$$

$$\forall\, p : Product;\ f : Feature;\ t_i : TreeFeature;\ t : \mathbb{F}_2\ TreeFeature \bullet$$
$$p \prec\!\!<_R mandatory(\ t_i\ )\ \Leftrightarrow\ p \prec\!\!<_\tau t_i\ \wedge$$
$$p \prec\!\!<_R optional(\ t_i\ )\ \Leftrightarrow\ (\ p \prec\!\!<_\tau t_i\ \vee\ p \cap \Phi_\tau\ t_i\ =\ \varnothing\ )\ \wedge$$
$$p \prec\!\!<_R one\_or\_more(\ t\ )\ \Leftrightarrow$$
$$(\ \forall\, t_j : t\ \bullet\ p \prec\!\!<_R optional(\ t_j\ )\ \wedge\ \exists\, t_k : t\ \bullet\ p \prec\!\!<_\tau t_k\ )\ \wedge$$
$$p \prec\!\!<_R only\_one(\ m\ )\ \Leftrightarrow$$
$$(\ \forall\, t_j : t\ \bullet\ p \prec\!\!<_R optional(\ t_j\ )\ \wedge\ \exists_1\, t_k : t\ \bullet\ p \prec\!\!<_\tau t_k\ )$$

Finally, the $\prec\!\!<$ predicate corresponding to CTCs (denoted as $\prec\!\!<_\chi$) is the following, where the usual semantics of logic implication and mutual exclusion are specified:

$$
\begin{array}{|l}
\_ \prec\!\!<_\chi \_ : Product \leftrightarrow CTC \qquad \text{[Instance of a BFM CTC]} \\
\hline
\forall\, p : Product;\ f_1, f_2 : Feature \bullet \\
\quad p \prec\!\!<_\chi requires(\, f_1, f_2\,) \ \Leftrightarrow\ (\, f_1 \notin p \ \vee\ f_2 \in p\,) \ \wedge \\
\quad p \prec\!\!<_\chi excludes(\, f_1, f_2\,) \ \Leftrightarrow\ (\, f_1 \notin p \ \vee\ f_2 \notin p\,)
\end{array}
$$

Once the abstract type $Model$ and the operations Φ (*features–in–a–model*) and $\prec\!\!<$ (*is–an–instance–of*) have been made concrete for the BFM notation in the CML of FLAME, all the analysis operations defined in the AFL (see Section 3) can be used without modification. Apart of this high level of reuse, BFM–specific operations like those commented in [Benavides et al., 2010] can also be defined in the CML of FLAME.

Chapter 5

Test–based validation of the FLAME framework

In order to validate the FLAME framework, its reference implementation, i.e. the Prolog animation of the formal specification, was tested using a fully automated approach following the IX commandment of formal methods proposed by Bowen and Hinchey [2012], which stresses the importance of testing when using formal methods. The goal of the test–based validation was twofold: (1) detecting faults in the reference implementation i.e., mismatches between the formal specification and its animation; and (2) detecting flaws in the Z specification, i.e. mismatches between what was intended to be specified and the actual specification. In the following sections the testing framework used, the validation setup and the main results obtained during the validation of the FLAME framework are presented.

5.1 The BeTTy framework

The test–based validation of the FLAME framework was performed using the automated approach presented by some of the authors in [Segura et al., 2011] and integrated into the open–source Java framework BeTTy [Segura et al., 2012], which enables the automated generation of test cases for the automated analysis of feature models.

In BeTTy, a test case is composed of a set of inputs (a feature model and some other optional parameters) and an expected output of the analysis operation under test. The key idea behind BeTTy is that the result of most analysis operations on feature models can be derived by inspecting theirs set of valid products adequately. Figure 5.1 depicts an example of how BeTTy works. The process starts with a trivial input feature model and its set of valid products. The model is then extended progressively by adding random feature relationships and cross–tree–constraints. The set of products is also updated at each step by using the so–called *metamorphic relationships* [Segura et al., 2011], i.e. the relationships between incremental changes in a feature model and their effects on the corresponding set of valid products. In Figure 5.1, the new feature relationships and cross–tree–constraints are depicted in gray, whereas the changes in the set of valid products caused by

their addition are depicted in bold face.

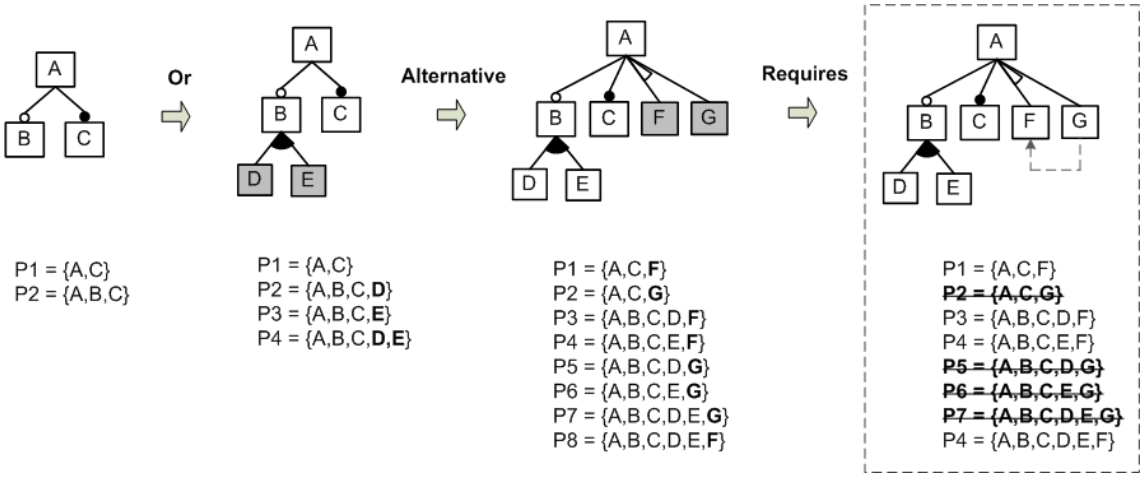

Figure 5.1: An example of random feature model generation using metamorphic relationships

Once a feature model with the desired properties is created, it is used as nontrivial input for the tests and its set of products is automatically inspected to compute the expected output of the analysis operations under tests. As an example, consider the model and set of products generated in Figure 5.1. The expected output of most of the analysis operations over the model can be obtained by simply answering simple questions such as:

- *Is the model void?* No, the set of products is not empty.
- *Is P = {`A`, `C`, `F` } a valid product?* Yes. It is included in the set of products.
- *How many different products represent the model?* There are 4 valid products.
- *What is the variability of the model?* There are 4 valid products and 7 features, so the variability is $4/(2^7 - 1) = 0.031 = 3.1\%$
- *What is the commonality of feature `B`?* Feature `B` is included in 3 out of the 4 products of the set. Therefore its commonality is 75%
- *Does the model contain any dead features?* Yes. Feature `G` is dead since it is not included in any of the products represented by the model.

The BeTTy framework has not only proved to be effective in detecting faults in actual tools for the automated analysis of feature models such as the FaMa framework [ISA Research Group, 2012] and SPLOT [Mendonca et al., 2009], but it has also shown to be much more effective than manually–designed test cases for the analysis of feature models, as described in [Segura et al., 2011].

5.2 Test–based validation setup

The formal specification and the reference implementation were developed in parallel, so it was possible to manually develop and run small tests as long as the analysis operations were specified and animated in Prolog. Although these tests were not developed systematically, they were quite useful for illustrating discussions among the authors and for detecting some problems related with 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, each of which is 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 feature model and an expected output. In the case of operations receiving other inputs apart from a feature model (e.g. *valid product*, which takes a product to be checked), these inputs were generated using a *partition equivalence strategy* [Beizer, 1990; Myers and Sandler, 2004], e.g. generating valid and non–valid products with equal probability.

For efficiency, feature models were generated with 10 features and 0–30% of cross–tree constraints with respect to the number of features. Previous works in testing by Segura et al. [2011] have shown that feature models with 10 features are complex enough to reveal faults effectively.

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 developed by Wielemaker [2012]. An example of such integration for the *number of valid products* operation is shown in Figure 5.2.

Once prepared, the tests were executed against the reference implementation and the results were checked. Whenever a fault was detected, the reference implementation 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.

5.2.3 Tests execution in FaMa

Finally, all the generated test cases were executed against the FaMa framework [ISA Research Group, 2012], a mature framework previously developed by some of the authors based on a informal description of the analysis operations and already known by the SPL community [Trinidad et al., 2008]. Using the Prolog animation of the formal specification in FLAME as a reference implementation, the FaMa framework was checked in order to prove their correctness and detect possible deviations from expected results. Due to the maturity of the FaMa framework, this step was also considered as a double–check for the reference implementation.

```
% SPL instances containing feature models 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 ).
```

Figure 5.2: Structure of a Prolog unitary test for the validation of the nop (*number of products*) operation

In order to avoid biased results, the systematic 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.

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 [Benavides et al., 2010]. A description of these faults and how they were fixed are next presented.

5.3.1 Variant and dead features

In [Benavides et al., 2010], *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, where 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 [Benavides et al., 2010] 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:

$$\begin{aligned}
&\forall\, spl : SPL \bullet \\
&\quad \langle\ \Phi_C\ spl,\ \Phi_V\ spl,\ \Phi_D\ spl\ \rangle \text{ partitions } spl.features
\end{aligned}$$

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 [Benavides et al., 2010] as:

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

and the test–based validation 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 by Fernandez-Amorós et al. [2009], it was clear that there was a mistake in the definition of homogeneity in [Benavides et al., 2010], so the formal specification and reference implementation of this operation in the FLAME framework was fixed in order to be compliant with their authors.

The *homogeneity bug* was a signal to review all numerical indicators where 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 [Benavides et al., 2010]. 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 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, *feature–model–dependent* atomic sets operation and the same notation–independent operation defined in the FLAME framework.

The concept of atomic set, as defined by Zhang et al. [2004a], Segura [2008] and as implemented in the FaMa framework, is defined only over feature models and it is based not on a formal specification but on an algorithm that merges parent features with their mandatory children features without considering cross–tree constraints. This concept of atomic sets allows a feature model to be reduced by replacing groups of features by the corresponding atomic sets in its feature tree, as shown in Figure 5.3 taken from [Benavides et al., 2010].

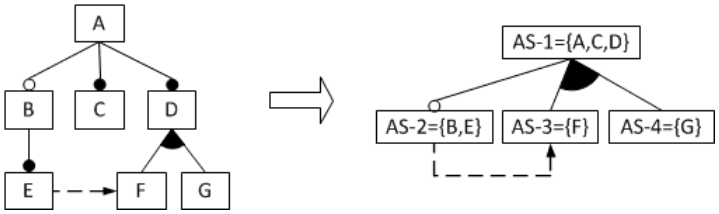

Figure 5.3: Feature model reduction applying atomic sets

In the FLAME framework, the concept of atomic sets is specified not structurally but semantically in a higher–level, notation–independent manner (see Section 3.4.5), so it is applicable not only to feature models but to any variability notation used as a characteristic model of an SPL. In FLAME, the atomics 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 these semantics, FLAME and FaMa frameworks produced the same atomic sets during the test–based validation except when cross–tree–constraints were relevant. For example, for the feature model in Figure 5.3, the atomic sets produced by both frameworks are the same. If a new cross–tree–constraint (`f requires e`) is introduced, the FaMa frameworks produces the same set of atomic sets, whereas the FLAME framework produces { `a, c, d` }, { `b, e, f` } and { `g` } (see Figure 5.4).

Notice that whereas the FaMa framework is able to reduce a feature model using its atomic

sets, the FLAME framework only computes the atomic sets, since is notation–independent. Notice also that those atomic sets computed by FLAME can be more accurate if cross–tree–constraints 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.

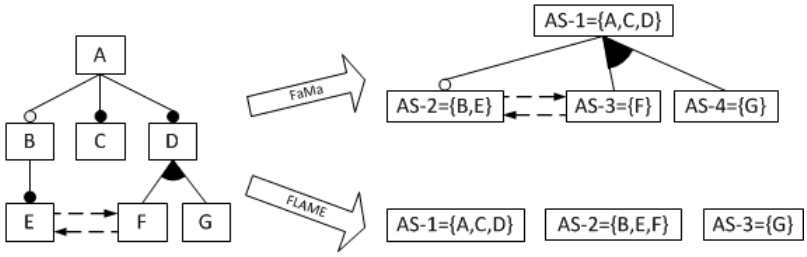

Figure 5.4: Different atomic sets from the FaMa and FLAME frameworks

5.3.4 Prolog toolkit for sets

In the early stages of the test–based validation, a fault in the Prolog toolkit developed for representing sets using lists (see the electronic appendix) was detected. The problem was related with 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 reference implementation could be systematically tested.

Chapter 6

Related work

In a recent literature review developed by some of the authors [Benavides et al., 2010], the formalization of analysis operations on feature models and their corresponding semantics were identified as a challenge. However, there are some proposals that already define formally, or at least with certain level of rigor, different analysis operations on feature models. Table 6.1 shows a summary of the related work found in the literature, including the FLAME framework in the last column.

	Zhang et al. [2004b]	Benavides et al. [2005]	von der Massen and Litcher [2005]	Sun et al. [2005]	Fan and Zhang [2006]	Gheyi et al. [2006]	Bachmeyer and Delugach [2007]	Schobbens et al. [2007]	Gheyi et al. [2008]	Mendonça et al. [2008]	Trinidad et al. [2008]	Zhang et al. [2008]	Fernandez-Amorós et al. [2009]	White et al. [2009]	**FLAME**
Abstraction	-	-	-	-	-	-	-	∼	-	-	-	-	-	-	**+**
Reference implementation	+	+	-	+	+	+	-	-	+	-	+	+	-	+	**+**
Test–based validation	-	-	-	∼	-	-	-	-	-	-	-	-	-	-	**+**
Number of operations	4	7	1	5	1	4	1	3	2	4	4	3	3	1	**18**

Table 6.1: Summary of the proposals reporting formalisation

The first row in Table 6.1 indicates if the proposal listed in the column is *abstract*, i.e. whether it specifies the semantics of the analysis operations at an abstract level without being coupled with any specific feature or variability model notation. In this sense, the FLAME framework, with all its analysis operations defined in the AFL (see Section 3), is a pioneer. Only Schobbens et al. [2007] proposes a sort of level of abstraction. In that work, a new feature model notation called VFD is defined and compared with other existing feature model 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 [Schobbens et al., 2007] but at a much more abstract level, not coupled to any specific feature model notation. Furthermore, the semantics of VFD or of any other variability model notations like OVM [Pohl et al., 2005]—which does not have a tree–like structure—, could be specified in the CML of the FLAME framework applying the systematic approach defined in Sections 3 and 4.

The second row in Table 6.1 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 [Benavides et al., 2005] a *constraint satisfaction problem* (CSP) solver is used to implement seven operations defined using CSP primitives. Likewise, Fan and Zhang [2006] 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–based animation as a reference implementation. Only Sun et al. [2005] follow a similar approach. In that work, Z is also used to specify the semantics of feature models and an implementation using Alloy [Jackson, 2012] is provided. However, the Z specification developed by Sun et al. [2005] is feature–model dependent, it cannot be extended to formalize other variability notations in a systematic way like FLAME, and the number of specified analysis operations is also lesser than in FLAME.

The third row in Table 6.1 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. [2005] follow a similar approach. In their work, Sun et al. [2005] perform a double validation. On the one hand, they develop 40 theorems and prove them using Z/EVES [Saaltink, 1997], 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 feature model to test it. In both cases, Sun et al. [2005] do not follow a systematic and automated approach as in the FLAME framework, so their validation process cannot be considered as thorough as the one performed for the FLAME framework.

Finally, the fourth row in Table 6.1 shows that FLAME has the highest number of analysis operations specified, all of them specified in a notation–independent manner. As commented at the end of Section 4, it is possible to specify feature–model dependent analysis operations in the CML of FLAME like *false optional features*, *conditionally dead features*, and others described by Benavides et al. [2010], although in this article only operations in the AFL of FLAME have been included in order to avoid an excessive length.

Chapter 7

Conclusions and future work

The first challenge identified by Benavides et al. [2010] in their survey on 20 years of feature models was to "formally describe all the operations of analysis [of feature models] and provide a formal framework for defining new operations". In this article, this challenge has been successfully faced.

The result, the FLAME framework, includes a formal specification in Z structured in two layers. In the former layer, the *abstract foundation layer*, it has been possible to specify analysis operations in an abstract and reusable way, making them applicable not only to feature models but to any variability notation. The latter layer, the *characteristic model layer*, is open for the specification of any variability notation following a systematic approach. An eclectic dialect of feature models (*basic feature model*) has been formalized in this article, but the same systematic approach could have been applied to other variability notations.

In order to support tool development, the FLAME framework also includes a reference implementation which is the result of the animation of the Z specification in Prolog. This reference implementation has been exhaustively tested with more than 18,000 test cases automatically generated using the BeTTy framework. As a result of the test–based validation of the Z specification, some analysis operations have been improved, a different semantics for *atomic sets* of features has been developed—which could lead to stronger model reductions in the future—, and the FaMa framework has been fixed and enhanced. In the FLAME framework, the novel combination of a formal specification, its corresponding animation and a automatic test–based validation has proved to be very effective.

The success in the development of the FLAME framework invites to apply the same approach to the formalization of other variability notations like OVM [Pohl et al., 2005] or other variants of feature models, for example those including cardinalities [Czarnecki et al., 2005], or those extending features with numerical information in the form of attributes and relationships among them [Benavides et al., 2005; Roos-Frantz et al., 2011]. Another interesting extension of FLAME is the formalization of notation–dependent analysis operations, some of them already mentioned in the survey by Benavides et al. [2010].

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 by Thüm et al. [2011] and Roos-Frantz [2012]. The study of how to include abstract features in FLAME leads to another appealing future work. Exploring other alternatives for specification animation seems also interesting, being Alloy and description logic the more likely candidates. The parallel development of the Z specification and its animation in Prolog—and the positive feedback between the two of them—enhanced them significantly. Using other implementation platforms could bring new synergies and increase the quality of the FLAME framework in the future.

Acknowledgments

The authors would like to thank José A. Galindo for his help implementing the BeTTy module for generating the tests in Prolog.

Bibliography

BACHMEYER, R. AND DELUGACH, H. 2007. A conceptual graph approach to feature modeling. In *Conceptual Structures: Knowledge Architectures for Smart Applications, 15th International Conference on Conceptual Structures, ICCS*. 179–191.

BATORY, D. 2005. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference*. Lecture Notes in Computer Sciences Series, vol. 3714. Springer–Verlag, 7–20.

BEIZER, B. 1990. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA.

BENAVIDES, D. 2007. On the automated analysis of software product lines using feature models. Ph.D. thesis, Dpto. Lenguajes y Sistemas Informáticos, ETS. Ingeniería Informática, Universidad de Sevilla.

BENAVIDES, D., RUIZ-CORTÉS, A., AND TRINIDAD, P. 2005. Automated reasoning on feature models. In *Advanced Information Systems Engineering: 17th International Conference, CAiSE*. Lecture Notes in Computer Sciences Series, vol. 3520. Springer–Verlag, 491–503.

BENAVIDES, D., SEGURA, S., AND RUIZ-CORTÉS, A. 2010. Automated Analysis of Feature Models 20 years Later: a Literature Review. *Information Systems 35,* 6.

BOWEN, J. P. AND HINCHEY, M. 2012. Ten commandments of formal methods... ten years on. In *Conquering Complexity*, M. Hinchey and L. Coyle, Eds. Springer London, 237–251.

CLEMENTS, P. AND NORTHROP, L. 2001. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison–Wesley.

CLOCKSIN, W. F. AND MELLISH, C. S. 2003. *Programming in Prolog: Using the ISO Standard* 5^{th} Ed. Springer–Verlag.

CZARNECKI, K., HELSEN, S., AND EISENECKER, U. 2005. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice 10,* 1, 7–29.

FAN, S. AND ZHANG, N. 2006. Feature model based on description logics. In *Knowledge-Based Intelligent Information and Engineering Systems, 10th International Conference, KES, Part II*. Lecture Notes in Computer Sciences Series, vol. 4252. Springer–Verlag.

FERNANDEZ-AMORÓS, D., HERADIO, R., AND CERRADA, J. 2009. Inferring information from feature diagrams to product line economic models. In *Proceedings of the Sofware Product Line Conference (SPLC'09)*.

GHEYI, R., MASSONI, T., AND BORBA, P. 2006. A theory for feature models in alloy. In *Proceedings of the ACM SIGSOFY First Alloy Workshop*. Portland, United States, 71–80.

GHEYI, R., MASSONI, T., AND BORBA, P. 2008. Algebraic laws for feature models. *Journal of Universal Computer Science 14,* 21, 3573–3591.

HEWITT, M., O'HALLORAN, C., AND SENNETT, C. 1997. Experiences with PiZA, an Animator for Z. In *ZUM'97: The Z Formal Specification Notation*, J. Bowen, M. Hinchey, and D. Till, Eds. LCNS Series, vol. 1212. Springer–Verlag, 35–51.

ISA RESEARCH GROUP. Accessed February 2012. FaMa Tool Suite. Available at `http://www.isa.us.es/fama/`.

ISO. 2002. Information technology – z formal specification notation – syntax, type system and semantics. iso/iec 13568:2002.

JACKSON, D. 2012. *Software Abstractions: Logic, Language, and Analysis* Revised Ed. MIT Press.

KANG, K., COHEN, S., HESS, J., NOVAK, W., AND PETERSON, S. 1990. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University. Nov.

KING, P. 1990. Printing Z and Object–Z LaTeX documents. Tech. rep., University of Queensland.

MENDONCA, M., BRANCO, M., AND COWAN, D. 2009. S.p.l.o.t. - software product lines online tools. In *Companion to the 24th ACM SIGPLAN International Conference on Object–Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*. Orlando, USA.

MENDONÇA, M., WASOWSKI, A., CZARNECKI, K., AND COWAN, D. 2008. Efficient compilation techniques for large scale feature models. In *Generative Programming and Component Engineering, 7th International Conference, GPCE , Proceedings*. 13–22.

MYERS, G. J. AND SANDLER, C. 2004. *The Art of Software Testing*. John Wiley & Sons.

POHL, K., BÖCKLE, G., , AND VAN DER LINDEN, F. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer–Verlag.

ROOS-FRANTZ, F. 2012. Automated analysis of software product lines with orthogonal variability models. Ph.D. thesis, Dpto. Lenguajes y Sistemas Informáticos, ETS. Ingeniería Informática, Universidad de Sevilla.

ROOS-FRANTZ, F., BENAVIDES, D., RUIZ-CORTÉS, A., HEUER, A., AND LAUENROTH, K. 2011. Quality-aware analysis in product line engineering with the orthogonal variability model. *Software Quality Journal*, 1–47.

SAALTINK, M. 1997. The z/eves system. In *ZUM '97: The Z Formal Specification Notation*, J. Bowen, M. Hinchey, and D. Till, Eds. Lecture Notes in Computer Science Series, vol. 1212. Springer Berlin / Heidelberg, 72–85.

SCHOBBENS, P., P. HEYMANS, J. T., AND BONTEMPS, Y. 2007. Generic semantics of feature diagrams. *Computer Networks 51,* 2, 456–479.

SEGURA, S. 2008. Automated analysis of feature models using atomic sets. In *First Workshop on Analyses of Software Product Lines (ASPL 2008). SPLC'08*. Limerick, Ireland, 201–207.

SEGURA, S., BENAVIDES, D., AND RUIZ-CORTÉS, A. 2011. Functional Testing of Feature Model Analysis Tools: A Test Suite. *IET Software 5,* 1.

SEGURA, S., GALINDO, J., BENAVIDES, D., PAREJO, J., AND RUIZ-CORTÉS, A. 2012. Betty: Benchmarking and testing on the automated analysis of feature models. In *Sixth International Workshop on Variability Modelling of Software–intensive Systems (VaMoS'12)*. Leipzig, Germany, 63–71. The BeTTy framework is available at `http://www.isa.us.es/betty`.

SEGURA, S., HIERONS, R. M., BENAVIDES, D., AND RUIZ-CORTÉS, A. 2011. Automated Metamorphic Testing on the Analyses of Feature Models. *Information and Software Technology 53,* 3.

SPIVEY, J. M. 1992. *The Z Notation: A Reference Manual* 2^{nd} Ed. Prentice–Hall.

SUN, J., ZHANG, H., LI, Y., AND WANG, H. 2005. Formal semantics and verification for feature modeling. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*.

THÜM, T., BATORY, D., AND KÄSTNER, C. 2009. Reasoning about edits to feature models. In *International Conference on Software Engineering*. 254–264.

THÜM, T., KASTNER, C., ERDWEG, S., AND SIEGMUND, N. 2011. Abstract features in feature modeling. In *Software Product Line Conference (SPLC'11), 2011 15th International*. IEEE, 191–200.

TRINIDAD, P., BENAVIDES, D., DURÁN, A., RUIZ-CORTÉS, A., AND TORO, M. 2008. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software 81,* 6, 883–896.

TRINIDAD, P., BENAVIDES, D., RUIZ-CORTÉS, A., SEGURA, S., AND JIMENEZ, A. 2008. Fama framework. In *12th Intl. Software Product Line Conference - Tool Demonstrations*. IEEE CS, 359 –359.

VON DER MASSEN, T. AND LITCHER, H. 2005. Determining the variation degree of feature models. In *Software Product Lines Conference*. Lecture Notes in Computer Sciences Series, vol. 3714. Springer–Verlag, 82–88.

WEST, M. M. AND EAGLESTONE, B. M. 1992. Software Development: Two Approaches to Animation of Z Specifications using Prolog. *Software Engineering Journal 7,* 4.

WHITE, J., BENAVIDES, D., SCHMIDT, D., TRINIDAD, P., DOUGHERTY, B., AND RUIZ-CORTÉS, A. 2010. Automated diagnosis of feature model configurations. *Journal of Systems and Software 83,* 7, 1094 – 1107.

WHITE, J., DOUGHTERY, B., SCHMIDT, D., AND BENAVIDES, D. 2009. Automated reasoning for multi-step software product-line configuration problems. In *Proceedings of the Sofware Product Line Conference*. 11–20.

WIELEMAKER, J. Accessed February 2012. Prolog unit tests. Available at `http://www.swi-prolog.org/pldoc/package/plunit.html`.

WOODCOCK, J. AND DAVIES, J. 1996. *Using Z: Specification, Refinement, and Proof*. Prentice–Hall.

ZHANG, W., YAN, H., ZHAO, H., AND JIN, Z. 2008. A bdd–based approach to verifying clone-enabled feature models' constraints and customization. In *High Confidence Software Reuse in Large Systems, 10th International Conference on Software Reuse, ICSR, Proceedings*. Lecture Notes in Computer Sciences Series, vol. 5030. Springer–Verlag, 186–199.

ZHANG, W., ZHAO, H., AND MEI, H. 2004a. A Propositional Logic-Based Method for Verification of Feature Models. In *Formal Methods and Software Engineering*, J. Davies, W. Schulte, and M. Barnett, Eds. LCNS Series, vol. 3308. Springer–Verlag, 115–130.

ZHANG, W., ZHAO, H., AND MEI, H. 2004b. A propositional logic-based method for verification of feature models. In *ICFEM 2004*, J. Davies, Ed. Lecture Notes in Computer Sciences Series, vol. 3308. Springer–Verlag, 115–130.

Appendix A

Prolog code of the reference implementation

This appendix contains the Prolog code corresponding to the reference implementation of the FLAME framework, which can also be downloaded from `http://www.isa.us.es/flame`. The structure of this appendix is as follows: Section A.1 shows a sample use of the FLAME framework; Sections A.2 and A.3 include the Prolog code corresponding to the *abstract foundation layer* (AFL) and the implementation of the semantics of the *basic feature model* (BFM) notation as a particular case of the *characteristic model layer* (CML) of the FLAME framework; Finally, Section A.4 includes the code corresponding to the set toolkit used in the FLAME framework.

A.1 Sample use of the FLAME framework

```
% ------------------------------------------------------------------------------
% File: FLAME_samples.pl
% Content: Sample uses of the FLAME framework
% Author: Amador Durán Toro
% Date: 05/03/2012
% ------------------------------------------------------------------------------

:- consult( 'afl/afl.pl'     ). % FLAME AFL layer
:- consult( 'cml/bfm/bfm.pl' ). % FLAME CML layer

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

spl_db( bad_spl,
  spl(
    [ f1, f2, f3 ],
    bfm( k_feature( f1, [
            mandatory( l_feature( f2 ) ),
            mandatory( l_feature( f4 ) ),
            optional( k_feature( f2, [
              only_one( [ l_feature( f1 ), l_feature( f5 ), l_feature( f4 ) ] )
            ] ) )
          ] ),
          [] % no cross-tree constraints
```

```
    )
  )
).

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

spl_db( atomic_sets_spl,
  spl(
    [ a, b, c, d, e, f, g ],
    bfm( k_feature( a, [
         optional( k_feature( b, [
             mandatory( l_feature( e ) )
         ] ) ),
         mandatory( l_feature( c ) ),
         mandatory( k_feature( d, [
             one_or_more( [ l_feature( f ), l_feature( g ) ] )
         ] ) )
    ] ),
    [ requires( e, f ),
      requires( f, e )
    ]
    )
  )
).

% 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 ),

  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,

homogeneity( SPL, H ),
write( 'Homogeneity = ' ), write( H ), nl,

variability( SPL, V ),
write( 'Variability = ' ), write( V ), nl,

partial_variability( SPL, PV ),
write( 'Partial variability = ' ), write( PV ), nl,

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

fail.
```

A.2 Abstract foundation layer of FLAME

```
% ------------------------------------------------------------------------------
% File: afl.pl
% Content: abstract foundation layer (AFL) package for FLAME
% Author: Amador Durán Toro
% Date: 24/03/2011
% ------------------------------------------------------------------------------

:- consult( 'sets.pl'         ). % set toolkit
:- consult( 'check_spl.pl'    ). % abstract SPL checking
:- consult( 'valid.pl'        ). % validity
:- consult( 'products.pl'     ). % products
:- consult( 'relations.pl'    ). % SPL relations
:- consult( 'filter.pl'       ). % filter
:- consult( 'features.pl'     ). % feature-related operations
:- consult( 'atomic_sets.pl' ). % atomic sets
:- consult( 'indicators.pl'   ). % numerical indicators
```

A.2.1 Abstract SPL checking

```
% ------------------------------------------------------------------------------
% File: check_spl.pl
% Content: SPL checks for FLAME
% Author: Amador Durán Toro
% Date: 24/03/2011
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% Z definitions (AFL)
%   +- SPL -------------------------------
%   | model : Model
%   | features : F1 Feature
%   +------------------
%   | features( model ) = features
%   | [ characteristic model checks ]
%   +---------------------------------------
%
% Prolog predicates
%   check_spl( spl( F, M ) )
%
% This predicate checks an SPL for internal correctness:
%   - if its set of features is not empty
%   - if its set of features is actually a set
%   - if its model is defined using only its set of features
%   - if its characteristic model is correct, using the abstract predicate
%     checkModel, that must redefined in the characteristic model layer.
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% check_spl( SPL )
% It checks an SPL for internal correctnes.
% ------------------------------------------------------------------------------

% No features
check_spl( spl( [], _ ) ) :-
   write( 'SPL is not OK!' ), nl,
   write( 'Empty feature set.' ), nl,
   fail.

% Features is not a set, i.e. it contains duplicates
```

```
check_spl( spl( F, _ ) ) :-
   not( is_set( F ) ),
   write( 'SPL is not OK!' ), nl,
   write( 'Duplicated features in its feature set.' ), nl,
   fail.

% Feature model uses unknown features
check_spl( spl( F, M ) ) :-
   features( M, FM ),
   not( subset( FM, F ) ),
   write( 'SPL is not OK!' ), nl,
   write( 'Feature model includes unknown features ' ),
   subtract( FM, F, U ),
   write( U ), nl,
   fail.

% Some features are not used in the feature model
check_spl( spl( F, M ) ) :-
   features( M, FM ),
   not( subset( F, FM ) ),
   write( 'SPL is not OK!' ), nl,
   write( 'Feature model does not use some features ' ),
   subtract( F, FM, U ),
   write( U ), nl,
   fail.

% Check characteristic model
check_spl( spl( F, M ) ) :-
   check_model( spl( F, M ) ), % abstract predicate
   fail.

% Success predicate (nothing else to be checked)
check_spl( spl( _, _ ) ) :-
  write( 'No (more) errors found in SPL.' ), nl.
```

A.2.2 Validity relations

```
% ------------------------------------------------------------------------------
% File: valid.pl
% Content: Validity relations of FLAME
% Author: Amador Durán Toro
% Date: 21/04/2011
% ------------------------------------------------------------------------------
% Update: fixed valid product - configuration; added full and partial
% Author: Amador Durán Toro
% Date: 09/02/2012
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% Z definitions
%   _valid_     : Product <-> SPL
%   _valid_p_c_ : Product <-> Configuration
%   _valid_c_   : Configuration <-> SPL
%
% Prolog predicates
%   valid( Product, spl( Features, Model ) )
%   valid_p_c( Product, Configuration )
%   valid_c( Configuration, spl( Features, Model ) )
%   why_not_valid( Product, spl( Features, Model ) )
%
% The first predicate checks if a product is valid for an SPL. The second does
% the same for a configuration. The third checks if a configuration is valid
```

```
% for an SPL. The fourth explains why a product is not valid.
% ------------------------------------------------------------------------

% ------------------------------------------------------------------------
% valid( Product, spl( Features, Model ) )
% It checks if a product is valid for an SPL using the abstract predicate
% instance_of, that must redefined in the characteristic model layer.
% ------------------------------------------------------------------------

valid( P, spl( F, M ) ) :-
  btrck_subset( P, F ), % backtrackable version of subset
  instance_of( P, M ).  % abstract predicate

% ------------------------------------------------------------------------
% valid_p_c( Product, Configuration( Selected, Removed ) )
% It checks if a product is valid for a configuration.
% ------------------------------------------------------------------------

valid_p_c( P, configuration( S, R ) ) :-
  subset( S, P ),
  intersection( R, P, [] ).

% ------------------------------------------------------------------------
% valid( Configuration( Selected, Removed ), spl( Features, Model ) )
% It checks if a configuration is valid for an SPL.
% ------------------------------------------------------------------------

valid_c( configuration( S, R ), spl( F, M ) ) :-
  union( S, R, U ),
  subset( U, F ),
  products( spl( F, M ), PRDS ),
  member( P, PRDS ),
  valid_p_c( P, configuration( S, R ) ).

% ------------------------------------------------------------------------
% full( Configuration( Selected, Removed ), spl( Features, Model ) )
% It checks if a configuration is full with respect to an SPL.
% ------------------------------------------------------------------------

full( configuration( S, R ), spl( F, M ) ) :-
  union( S, R, U ),
  equal_set( U, F ).

% ------------------------------------------------------------------------
% partial( Configuration( Selected, Removed ), spl( Features, Model ) )
% It checks if a configuration is partial with respect to an SPL.
% ------------------------------------------------------------------------

partial( configuration( S, R ), spl( F, M ) ) :-
  union( S, R, U ),
  proper_subset( U, F ).

% ------------------------------------------------------------------------
% why_not_valid( Product, spl( Features, Model ) )
% It explains why a product is not valid.
% ------------------------------------------------------------------------

% The product is an empty set
why_not_valid( [], _ ) :-
   !, write( 'The product contains no features.' ), nl.

% The product is not a nonempty set, i.e. contains duplicated features
why_not_valid( P, _ ) :-
```

```
   not( is_set( P ) ), !,
   write( 'Product ' ),
   writeq( P ),
   write( ' contains duplicated features.' ), nl.

% The product is not a subset of the feature set, i.e. contains unknonw features
why_not_valid( P, spl( F, _ ) ) :-
   not( subset( P, F ) ), !,
   write( 'Product ' ),
   writeq( P ),
   write( ' contains unknown features.' ), nl.

% The product is not an instance of the model
why_not_valid( P, spl( _, M ) ) :-
   not( instance_of( P, M ) ), !,
   write( 'Product ' ),
   writeq( P ),
   write( ' is not an instance of the model.' ), nl.

% The product is valid
why_not_valid( P, _ ) :-
  write( 'PRODUCT ' ),
  writeq( P ),
  write( ' IS VALID!' ), nl.
```

A.2.3 Valid products

```
% ------------------------------------------------------------------------------
% File: products.pl
% Content: Product functions and relations of FLAME
% Author: Amador Durán Toro
% Date: 02/04/2011
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% Z definitions
%   products : SPL -> F Product
%   nop : SPL -> N
%   void: P SPL
%
% Prolog predicates (public)
%   products( spl( Features, Model ), [ Products ] )
%   products_verbose( spl( Features, Model ), [ Products ] )
%   nop( spl( Features, Model ), N )
%   void( spl( Features, Model )
%
% These predicates return the set of valid products of an SPL, the number
% of products, i.e. the cardinality of the set of valid products, and check if
% an SPL is void.
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% products( spl( Features, Model ), Products )
% It computes the set of products of an SPL using the valid predicate.
% ------------------------------------------------------------------------------

products( spl( F, M ), PRDS ) :-
  findall( P, valid( P, spl( F, M ) ), PRDS ).

% ------------------------------------------------------------------------------
% products_verbose( spl( Features, Model ), Products )
% Same as products but showing progress. It does not use findall but a classic
```

```
% accumulator-based auxiliary predicate.
% ------------------------------------------------------------------------------

products_verbose( spl( F, M ), PRDS ) :-
  write( 'Computing products...' ), nl,
    power_set( F, PWR ),
    select( [], PWR, PWR1 ), % take the emptyset away
  length( PWR1, NOPP ),
  write( 'Checking ' ),
  writeq( NOPP ),
  write( ' potential products:' ), nl,
    products_aux( PWR1, spl( F, M ), [], PRDS ),
  length( PRDS, NOP ),
  write( 'There are ' ),
  writeq( NOP ),
  write( ' products in the SPL.' ), nl.

products_aux( [], _, PRDS, PRDS ).

products_aux( [ Pi | P ], spl( F, M ), ACC1, PRDS ) :-
  valid( Pi, spl( F, M ) ), !,
  why_not_valid( Pi, spl( F, M ) ),
  append( [ Pi ], ACC1, ACC2 ),
  products_aux( P, spl( F, M ), ACC2, PRDS ).

products_aux( [ Pi | P ], spl( F, M ), ACC, PRDS ) :-
  %why_not_valid( Pi, spl( F, M ) ),
  products_aux( P, spl( F, M ), ACC, PRDS ).

% ------------------------------------------------------------------------------
% nop( spl( Features, Model ), NoP )
% It computes the number of products of an SPL.
% ------------------------------------------------------------------------------

nop( spl( F, M ), NOP ) :-
  products( spl( F, M ), PRDS ),
  length( PRDS, NOP ).

% ------------------------------------------------------------------------------
% void( spl( Features, Model ) )
% It computes if the SPL is void.
% ------------------------------------------------------------------------------

void( spl( F, M ) ) :-
  products( spl( F, M ), [] ).
```

A.2.4 Filtering

```
% ------------------------------------------------------------------------------
% File: filter.pl
% Content: SPL filtering of FLAME
% Author: Amador Durán Toro
% Date: 21/04/2011
% ------------------------------------------------------------------------------
% Update: cut eliminated from filter
% Author: Amador Durán Toro
% Date: 13/02/2012
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% Z definitions
%   filter : SPL x Configuration -> F Product
```

```
%
% Prolog predicates
%   filter( spl( Features, Model ), configuration( S, R ), Result )
%
% This predicate returns the set of products of the SPL according to the
% configuration.
% -----------------------------------------------------------------------------

% -----------------------------------------------------------------------------
% filter( spl( Features, Model ), configuration( S, R ), Result )
% It returns the set of products of the SPL according to the configuration.
% -----------------------------------------------------------------------------

filter( spl( F, M ), configuration( S, R ), RESULT ) :-
  products( spl( F, M ), PRDS ),
  findall( P, filtered( P, configuration( S, R ), PRDS ), RESULT ).

filtered( P, configuration( S, R ), PRDS ) :-
  member( P, PRDS ),
  valid_p_c( P, configuration( S, R ) ).
```

A.2.5 SPL relations

```
% -----------------------------------------------------------------------------
% File: relations.pl
% Content: SPL relations of FLAME
% Author: Amador Durán Toro
% Date: 07/07/2011
% -----------------------------------------------------------------------------

% -----------------------------------------------------------------------------
% Z definitions
%   _equivalent_ : SPL <-> SPL
%   _generalization_ : SPL <-> SPL
%   _specialization_ : SPL <-> SPL
%   _arbitrary_edit_ : SPL <-> SPL
%
% Prolog predicates
%   equivalent( spl( F1, M1 ), spl( F2, M2 ) )
%   specialization( spl( F1, M1 ), spl( F2, M2 ) )
%   generalization( spl( F1, M1 ), spl( F2, M2 ) )
%   arbitrary_edit( spl( F1, M1 ), spl( F2, M2 ) )
%
% These predicates check if both SPLs are equivalent, a generalization/
% specialization of each other or an arbitrary edit.
% -----------------------------------------------------------------------------

% -----------------------------------------------------------------------------
% equivalent( spl( F1, M1 ), spl( F2, M2 ) )
% It checks if both SPLs are equivalent.
% -----------------------------------------------------------------------------

equivalent( spl( F1, M1 ), spl( F2, M2 ) ) :-
  products( spl( F1, M1 ), PRDS1 ),
  products( spl( F2, M2 ), PRDS2 ),
  equal_set( PRDS1, PRDS2 ).

% -----------------------------------------------------------------------------
% specialization( spl( F1, M1 ), spl( F2, M2 ) )
% It checks if the first SPL is a specialization of the second.
% -----------------------------------------------------------------------------
```

```
specialization( spl( F1, M1 ), spl( F2, M2 ) ) :-
  products( spl( F1, M1 ), PRDS1 ),
  products( spl( F2, M2 ), PRDS2 ),
  proper_subset( PRDS1, PRDS2 ).

% ------------------------------------------------------------------------------
% generalization( spl( F1, M1 ), spl( F2, M2 ) )
% It checks if the first SPL is a generalization of the second.
% ------------------------------------------------------------------------------

generalization( spl( F1, M1 ), spl( F2, M2 ) ) :-
  specialization( spl( F2, M2 ), spl( F1, M1 ) ).

% ------------------------------------------------------------------------------
% arbitrary_edit( spl( F1, M1 ), spl( F2, M2 ) )
% It checks if both SPLs are an arbitrary edit of each other.
% ------------------------------------------------------------------------------

arbitrary_edit( spl( F1, M1 ), spl( F2, M2 ) ) :-
  not( equivalent( spl( F1, M1 ), spl( F2, M2 ) ) ),
  not( specialization( spl( F1, M1 ), spl( F2, M2 ) ) ),
  not( generalization( spl( F1, M1 ), spl( F2, M2 ) ) ).
```

A.2.6 Feature–related operations

```
% ------------------------------------------------------------------------------
% File: features.pl
% Content: Feature functions of FLAME
% Author: Amador Durán Toro
% Date: 07/04/2011
% ------------------------------------------------------------------------------
% Update: variant features semantics fixed
% Author: Amador Durán Toro
% Date: 03/06/2011
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% Z definitions
%   coreFeatures : SPL -> F Feature
%   variantFeatures : SPL -> F Feature
%   deadFeatures : SPL -> F Feature
%   uniqueFeatures : SPL -> F Feature
%
% Prolog predicates
%   core_features( SPL, [ Feature ] )
%   variant_features( SPL, [ Feature ] )
%   dead_features( SPL, [ Feature ] )
%   unique_features( SPL, [ Feature ] )
%
% These predicates compute different sets of features of an SPL.
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% core_features( SPL, [ Feature ] )
% It computes the set of core features of an SPL
% ------------------------------------------------------------------------------

core_features( spl( F, M ), C ) :-
  products( spl( F, M ), PRDS ),
  intersection( PRDS, C ).

% ------------------------------------------------------------------------------
```

```
% variant_features( SPL, [ Feature ] )
% It computes the set of variant features of an SPL
% ------------------------------------------------------------------------------

variant_features( spl( F, M ), V ) :-
  core_features( spl( F, M ), C ),
  dead_features( spl( F, M ), D ),
  subtract( F, C, VAUX ),
  subtract( VAUX, D, V ).

% ------------------------------------------------------------------------------
% dead_features( SPL, [ Feature ] )
% It computes the set of dead features of an SPL
% ------------------------------------------------------------------------------

dead_features( spl( F, M ), D ) :-
  products( spl( F, M ), PRDS ),
  union( PRDS, U ),
  subtract( F, U, D ).

% ------------------------------------------------------------------------------
% unique_features( SPL, [ Feature ] )
% It computes the set of unique features of an SPL
% ------------------------------------------------------------------------------

unique_features( spl( F, M ), RESULT ) :-
  products( spl( F, M ), PRDS ),
  findall( Fi, unique( Fi, F, PRDS ), RESULT ).

unique( Fi, F, PRDS ) :-
  member( Fi, F ),
  findall( P, contains( P, Fi, PRDS ), RESULT ),
  length( RESULT, 1 ).

contains( P, F, PRDS ) :-
  member( P, PRDS ),
  member( F, P ).
```

A.2.7 Atomic sets

```
% ------------------------------------------------------------------------------
% File: atomic_sets.pl
% Content: Atomic sets of FLAME
% Author: Amador Durán Toro
% Date: 07/04/2011
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% Z definitions
%   atomicSets : SPL -> F F Feature
%
% Prolog predicates
%   atomic_sets( SPL, [ [ Feature ] ] )
%   potential_atomic_sets( SPL, [ [ Feature ] ] )
%   atomic_set( SPL, [ Feature ] )
%
% These predicates compute the atomic sets of an SPL.
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% atomic_sets( SPL, [ [ Feature ] ] )
% It computes the (maximal) atomic sets of an SPL
```

```
% ----------------------------------------------------------------------------

atomic_sets( spl( F, M ), ATOMS ) :-
  potential_atomic_sets( spl( F, M ), POTATOMS ),
  findall( A, maximal( A, POTATOMS ), ATOMS ).

% ----------------------------------------------------------------------------
% potential_atomic_sets( SPL, [ [ Feature ] ] )
% It computes the potential atomic sets of an SPL using the auxiliary predicate
% atomicSet.
% ----------------------------------------------------------------------------

potential_atomic_sets( spl( F, M ), POTATOMS ) :-
  products( spl( F, M ), PRDS ),
  findall( A0, potential_atomic_set( A0, F, PRDS ), POTATOMS ).

potential_atomic_set( A0, F, PRDS ) :-
  btrck_subset( A0, F ), % backtrackable version of subset
  A0 \= [],              % atomic sets are nonempty
  forall( member( P, PRDS ), subset_or_disjoint( A0, P ) ).
```

A.2.8 Numerical indicators

```
% ----------------------------------------------------------------------------
% File: indicators.pl
% Content: Numerical indicators of FLAME
% Author: Amador Durán Toro
% Date: 10/04/2011
% ----------------------------------------------------------------------------
% Update: homogeneity, commonality and partial variability fixed
% Author: Amador Durán Toro
% Date: 07/06/2011
% ----------------------------------------------------------------------------
% Update: homogeneity re-fixed
% Author: Amador Durán Toro
% Date: 13/06/2011
% ----------------------------------------------------------------------------

% ----------------------------------------------------------------------------
% Z definitions
%   homogeneity : SPL -> R
%   commonality : SPL x Configuration -> R
%   variability : SPL -> R
%   pvariability : SPL -> R
%
% Prolog predicates
%   homogeneity( SPL, H )
%   commonality( SPL, CONF, C )
%   variability( SPL, V )
%   partial_variability( SPL, PV )
%
% These predicates compute several measures of an SPL.
% ----------------------------------------------------------------------------

% ----------------------------------------------------------------------------
% homogeneity( SPL, P )
% It computes the homogeneity of an SPL.
% ----------------------------------------------------------------------------

homogeneity( spl( F, M ), H ) :-
  unique_features( spl( F, M ), UF ),
  length( UF, NOUF ),
```

```
  length( F, NOF ),
  H is 1 - ( NOUF / NOF ).

% ------------------------------------------------------------------------------
% commonality( SPL, CONF, C )
% It computes the commonality factor of a configuration in an SPL.
% ------------------------------------------------------------------------------

commonality( spl( F, M ), configuration( _, _ ), 0 ) :-
  void( spl( F, M ) ).

commonality( spl( F, M ), configuration( S, R ), C ) :-
  filter( spl( F, M ), configuration( S, R ), FILTERED ),
  nop( spl( F, M ), NOP ),
  length( FILTERED, NOFP ),
  C is NOFP / NOP.

% ------------------------------------------------------------------------------
% variability( SPL, V )
% It computes the (total) variability of an SPL
% ------------------------------------------------------------------------------

variability( spl( F, M ), V ) :-
  nop( spl( F, M ), N ),
  length( F, NOF ),
  V is N / ( 2**NOF - 1 ).

% ------------------------------------------------------------------------------
% partial_variability( SPL, V )
% It computes the partial variability of an SPL
% ------------------------------------------------------------------------------

partial_variability( spl( F, M ), 0 ) :-
  variant_features( spl( F, M ), [] ).

partial_variability( spl( F, M ), PV ) :-
  nop( spl( F, M ), N ),
  variant_features( spl( F, M ), VF ),
  length( VF, NOVF ),
  PV is N / ( 2**NOVF - 1 ).

% ------------------------------------------------------------------------------
% potential( SPL, P )
% It computes the number of potential products of an SPL
% ------------------------------------------------------------------------------

%potential( spl( F, _ ), P ) :-
%  length( F, NOF ),
%  P is 2**NOF - 1.
```

A.3 Characteristic model layer of FLAME

```
% ------------------------------------------------------------------------------
% File: bfm.pl
% Content: CML/BFM package for FLAME
% Author: Amador Durán Toro
% Date: 24/03/2011
% ------------------------------------------------------------------------------

:- consult( 'helpers.pl'     ). % helper functions
:- consult( 'check_model.pl' ). % BFM model checking
:- consult( 'features.pl'    ). % features-in-a-model function
:- consult( 'instance.pl'    ). % is-instance-of relation
```

A.3.1 Helper functions

```
% ------------------------------------------------------------------------------
% File: helpers.pl
% Content: BFM helper functions for FLAME
% Author: Amador Durán Toro
% Date: 23/04/2011
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% Z definitions
%   children : Relationship -> F1 TreeFeature
%   feature_count_T : Feature x TreeModel -> N
%   feature_count_R : Feature x Relationship -> N
%
% Prolog predicates
%   children( Relationship, [ TreeFeature ] )
%   feature_count_t( Feature, TreeModel, N )
%   feature_count_r( Feature, Relationship, N )
%
% The first predicate returns the children TreeFeatures of a Relationship.
% The two last predicates return the number of times a feature appears in a
% TreeModel or in a Relationship.
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% children( Relationship, [ TreeFeature ] )
% It returns the children TreeFeatures of a Relationship.
% ------------------------------------------------------------------------------

children( mandatory( Ti ), CHLD ) :-
  children( optional( Ti ), CHLD ).

children( optional( Ti ), [ Ti ] ).

children( one_or_more( T ), CHLD ) :-
  children( only_one( T ), CHLD ).

children( only_one( T ), T ).

% ------------------------------------------------------------------------------
% feature_count_t( Feature, TreeModel, N )
% It computes the number of times a feature appears in a TreeModel.
% ------------------------------------------------------------------------------

feature_count_t( F, l_feature( F ), 1 ).
```

```
feature_count_t( F1, l_feature( F2 ), 0 ) :-
  F1 \= F2.

feature_count_t( F, k_feature( F, RELS ), N ) :-
  for_each_sum( feature_count_r, F, RELS, NR ),
  N is NR + 1.

feature_count_t( F1, k_feature( F2, RELS ), N ) :-
  F1 \= F2,
  for_each_sum( feature_count_r, F1, RELS, N ).

% ------------------------------------------------------------------------------
% feature_count_r( Feature, Relationship, N )
% It computes the number of times a feature appears in a Relationship.
% ------------------------------------------------------------------------------

feature_count_r( F, R, N ) :-
  children( R, CHLD ),
  for_each_sum( feature_count_t, F, CHLD, N ).
```

A.3.2 Specific BFM checking

```
% ------------------------------------------------------------------------------
% File: check_model.pl
% Content: BFM checks for FLAME
% Author: Amador Durán Toro
% Date: 23/04/2011
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% Z definitions (CML/BFM)
%   +- SPL -----------------------------------
%   | model : Model
%   | features : F1 Feature
%   +------------------
%   | features( model ) = features
%   | forAll f : features @
%   |   feature_count( f, tree model ) = 1
%   +-----------------------------------------
%
% Prolog predicates
%   check_model( spl( F, M ) )
%   check_tree( F, M )
%
% This predicate checks a BFM-based SPL for internal correctness:
%   - if its tree model is a tree
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% Model is not a tree
% ------------------------------------------------------------------------------

check_model( spl( _, bfm( TREE, CTC ) ) ) :-
   features( bfm( TREE, CTC ), F2 ),
   check_tree( F2, TREE ),
   fail.

% ------------------------------------------------------------------------------
% check_tree( [ F ], TREE )
% It checks if the features appear only once in the tree model using the
% check_feature_count auxiliary predicate.
% ------------------------------------------------------------------------------
```

```
check_tree( [], _ ).

check_tree( [ Fi | F ], TREE ) :-
   check_feature_count( Fi, TREE ),
   check_tree( F, TREE ).

check_feature_count( F, TREE ) :-
   feature_count_t( F, TREE, N ),
   N > 1, !,
   write( 'SPL is not OK!' ), nl,
   write( 'Feature ' ),
   write( F ),
   write( ' appears ' ), write( N ), write( ' times in the SPL model.' ), nl.

check_feature_count( _, _ ).
```

A.3.3 Features–in–a–model function

```
% ------------------------------------------------------------------------------
% File: features.pl
% Content: BFM features-in-a-model function for FLAME
% Author: Amador Durán Toro
% Date: 23/04/2011
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% Z definitions
%   features : Model -> F Feature
%   tfeatures : TreeFeature -> F Feature
%   rfeatures : Relationship -> F Feature
%   xfeatures : CTC -> F Feature
%
% Prolog predicates
%   features( Model, [ Feature ] )
%   t_features( TreeFeature, [ Feature ] )
%   r_features( Relationship, [ Feature ] )
%   x_features( CTC, [ Feature ] )
%
% These predicates return a set with all the features used in a BFM model.
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% features( Model, [ Feature ] )
% It computes the features in a BMF characteristic model
% ------------------------------------------------------------------------------

features( bfm( TREE, CTC ), F ) :-
  t_features( TREE, F1 ),
  for_each_union( x_features, CTC, F2 ),
  union( F1, F2, F ).

% ------------------------------------------------------------------------------
% t_features( TreeFeature, [ Feature ] )
% It computes the features in a TreeFeature.
% ------------------------------------------------------------------------------

t_features( l_feature( Fi ), [ Fi ] ).

t_features( k_feature( Fi, R ), F ) :-
  for_each_union( r_features, R, FR ),
  union( [ Fi ], FR, F ).
```

```
% ------------------------------------------------------------------------------
% r_features( Relationship, [ Feature ] )
% It computes the features in a Relationship.
% ------------------------------------------------------------------------------

r_features( R, F ) :-
  children( R, CHLD ),
  for_each_union( t_features, CHLD, F ).

% ------------------------------------------------------------------------------
% x_features( CTC, [ Feature ] )
% It computes the features in a CTC.
% ------------------------------------------------------------------------------

x_features( requires( F1, F2 ), F ) :-
  x_features( excludes( F1, F2 ), F ).

x_features( excludes( F1, F2 ), F ) :-
  union( [ F1 ], [ F2 ], F ).
```

A.3.4 Is–instance–of relation

```
% ------------------------------------------------------------------------------
% File: instance.pl
% Content: BFM is-instance-of relation for FLAME
% Author: Amador Durán Toro
% Date: 22/04/2011
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% NOTE: instance is a built-in predicate of SWI-Prolog that cannot redefined
% ------------------------------------------------------------------------------
%
% Z definitions
%   _instance_ : Product <-> Model
%   _tinstance_ : Product <-> TreeFeature
%   _rinstance_ : Product <-> Relationship
%   _xinstance_ : Product <-> CTC
%
% Prolog predicates
%   instance_of( Product, Model )
%   instance_of_t( Product, TreeFeature )
%   instance_of_r( Product, Relationship )
%   instance_of_x( Product, CTC )
%   instance_or_disjoint( Product, TreeFeature, NOI )
%   instance_or_disjoint( Product, [ TreeFeature ], NOI )
%
% The first predicates check if a product is an instance of a BFM model, a tree,
% a relationship, and a CTC. The last one is a helper predicate that checks if a
% product is an instance of or does not contain any feature of a list of trees.
% It also computes the number of times the product is an instance of the treee
% (NOI = Number Of Instances).
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% instance_of( Product, Model )
% It computes if the product is an instance of a BFM model.
% ------------------------------------------------------------------------------

instance_of( P, bfm( TREE, CTC ) ) :-
  instance_of_t( P, TREE ), !,
  forall( member( CTCi, CTC ), instance_of_x( P, CTCi ) ).

% ------------------------------------------------------------------------------
% instance_of_t( Product, TreeFeature )
% It computes if the product is an instance of a TreeFeature.
% ------------------------------------------------------------------------------

instance_of_t( P, l_feature( F ) ) :-
  member( F, P ).

instance_of_t( P, k_feature( F, R ) ) :-
  member( F, P ),
  forall( member( Ri, R ), instance_of_r( P, Ri ) ).

% ------------------------------------------------------------------------------
% instance_of_r( Product, Relationship )
% It computes if the product is an instance of a Relationship.
% ------------------------------------------------------------------------------

instance_of_r( P, mandatory( Ti ) ) :-
  instance_of_t( P, Ti ).
```

```
instance_of_r( P, optional( Ti ) ) :-
  instance_or_disjoint( P, Ti, _ ).

instance_of_r( P, one_or_more( T ) ) :-
  instance_or_disjoint( P, T, NOI ),
  NOI >= 1.

instance_of_r( P, only_one( T ) ) :-
  instance_or_disjoint( P, T, NOI ),
  NOI =:= 1.

% ------------------------------------------------------------------------------
% instance_of_x( Product, CTC )
% It computes if the product is an instance of a CTC.
% ------------------------------------------------------------------------------

instance_of_x( P, requires( F1, F2 ) ) :-
  not( member( F1, P ) )
  ;
  member( F2, P ).

instance_of_x( P, excludes( F1, F2 ) ) :-
  not( member( F1, P ) )
  ;
  not( member( F2, P ) ).

% ------------------------------------------------------------------------------
% instance_or_disjoint( Product, [ TreeFeature ], NOI )
% instance_or_disjoint( Product, TreeFeature, NOI )
%
% It checks if the product is an instance of the TreeFeature or if none of the
% features in the product appear in the tree. It also computes how many times
% the product is an instance of the tree (NOI = Number Of Instances).
% ------------------------------------------------------------------------------

instance_or_disjoint( _, [], NOI ) :-
  NOI is 0.

instance_or_disjoint( P, [ TFi | TF ], NOI ) :-
  instance_or_disjoint( P, TFi, NOI1 ),
  instance_or_disjoint( P, TF, NOI2 ),
  NOI is NOI1 + NOI2, !.

instance_or_disjoint( P, TFi, NOI ) :-
  instance_of_t( P, TFi ),
  NOI is 1.

instance_or_disjoint( P, TFi, NOI ) :-
  t_features( TFi, F ),
  intersection( P, F, [] ),
  NOI is 0.
```

A.4 Set toolkit of FLAME

```
% ------------------------------------------------------------------------------
% File: sets.pl
% Content: Set toolkit for FLAME
% Author: Amador Durán Toro
% Date: 24/04/2011
% ------------------------------------------------------------------------------
% Update: fixed powerset_subset by sorting both sets before comparing
% Author: Amador Durán Toro
% Date: 22/05/2011
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% Prolog predicates (public)
%   equal_set( [ Set ], [ Set ] )
%   equal_superset( [ [ Set ] ], [ [ Set ] ] )
%   union( [ [ Set ] ], [ Set ] )
%   intersection( [ [ Set ] ], [ Set ] )
%   maximal( [ Set ], [ [ Set ] ] )
%   proper_subset( [ Subset ], [ Set ] )
%   subset_or_disjoint( [ Subset ], [ Set ] )
%   power_set( [ Set ], [ [ Powerset ] ] )
%   btrck_subset( Subset, Set )
%   powerset_subset( [ Subset ], [ Set ] )
%   bratko_subset( [ Subset ], [ Set ] )
%   safe_sort( [ Set ], [ Set ] )
%   for_each_union( Goal, [ Set ], [ Set ] )
%   for_each_sum( Goal, [ Set ], N )
%   for_each_sum( Goal, K, [ Set ], N )
%   for_each_sort( [ [ Set ] ], [ [ Set ] ] )
%
% Some of these set-related predicates are implemented in other versions of
% Prolog, but not in SWI-Prolog.
% ------------------------------------------------------------------------------

% ------------------------------------------------------------------------------
% equal_set( [ Set ], [ Set ] )
% Predicate that checks set equality as mutual inclusion.
% ------------------------------------------------------------------------------

equal_set( Set1, Set2 ) :-
  subset( Set1, Set2 ),
  subset( Set2, Set1 ).

% ------------------------------------------------------------------------------
% equal_superset( [ [ Set ] ], [ [ Set ] ] )
% Predicate that checks superset (set of sets) equality.
% ------------------------------------------------------------------------------

equal_superset( S1, S2 ) :-
  for_each_sort( S1, SS1 ),
  for_each_sort( S2, SS2 ),
  equal_set( SS1, SS2 ).

% ------------------------------------------------------------------------------
% union( [ [ Set ] ], [ Set ] )
% It computes the distributed union over a set of sets.
% ------------------------------------------------------------------------------

union( Superset, Union ) :-
  flatten( Superset, Flatten ),
  list_to_set( Flatten, Union ).
```

```
% ------------------------------------------------------------------------------
% intersection( [ [ Set ] ], [ Set ] )
% It computes the distributed intersection over a set of sets using the
% auxiliary predicate dinter.
% ------------------------------------------------------------------------------

intersection( [], [] ).

intersection( [ S1 | Rest ], Intersection ) :-
  dinter( [ S1 | Rest ], S1, Intersection ).

dinter( [], Result, Result ).

dinter( [ S1 | Rest ], PreviousResult, Result ) :-
  intersection( S1, PreviousResult, Aux ),
  dinter( Rest, Aux, Result ).

% ------------------------------------------------------------------------------
% maximal( [ Set ], [ [ Set ] ] )
% This backtrackable predicate checks if the first set is a maximal subset of
% the set of sets passed as the second argument.
% ------------------------------------------------------------------------------

maximal( Si, S ) :-
  member( Si, S ),
  forall( member( Sj, S ), not( proper_subset( Si, Sj ) ) ).

% ------------------------------------------------------------------------------
% proper_subset( [ Subset ], [ Set ] )
% It checks if the first set is a proper subset of the second. In order to
% ignore if both sets are sorted, lengths are compared to determine if they
% are not the same set.
% ------------------------------------------------------------------------------

proper_subset( Si, Sj ) :-
  length( Si, L1 ),
  length( Sj, L2 ),
  L1 \= L2,
  subset( Si, Sj ).

% ------------------------------------------------------------------------------
% subset_or_disjoint( [ Subset ], [ Set ] )
% It checks if the first set is a subset of the second or if they are disjoint.
% ------------------------------------------------------------------------------

subset_or_disjoint( A, P ) :-
  subset( A, P )
  ;
  intersection( A, P, [] ).

% ------------------------------------------------------------------------------
% power_set( [ Set ], [ [ Powerset ] ] )
% Powerset predicate. It simply reverses the set before computing its powerset
% in order to generate more "human-like" results (first element first, etc.).
% ------------------------------------------------------------------------------

power_set( Set, Powerset ) :-
  reverse( Set, SetReverse ), !,
  power_set_aux( SetReverse, [[]], Powerset ), !.

% ------------------------------------------------------------------------------
% power_set_aux( [ Set ], [ Acc ], [ Powerset ] )
```

```
% Helper predicates borrowed from "Simply Logical" from Peter Flach (2005)
% http://www.cs.bris.ac.uk/Teaching/Resources/COMS30106/slides/SLchapter3.pdf
% ------------------------------------------------------------------------------

power_set_aux( [], Powerset, Powerset ).

power_set_aux( [ H | T ], Acc, PowerSet ):-
  extend_pset( H, Acc, Acc1 ),
  power_set_aux( T, Acc1, PowerSet ).

extend_pset( _, [], []).

extend_pset( H, [ List | MoreLists ],[ List, [ H | List ] | More ] ):-
  extend_pset( H, MoreLists, More ).

% ------------------------------------------------------------------------------
% btrck_subset( Subset, Set )
% It checks if the first set is a subset of the second. It is simple a delegate
% for actual backtrackable subset predicates. It can be configured to use
% powerset_subset (nice results) or bratko_subset (faster).
% ------------------------------------------------------------------------------

btrck_subset( Subset, Set ) :- % choose one implementation
  powerset_subset( Subset, Set ).
%  bratko_subset( Subset, Set ).

% ------------------------------------------------------------------------------
% powerset_subset( [ Subset ], [ Set ] )
% Backtrackable powerset-based subset predicate.
%
% Note: it doesn't work if both sets are not in the same order; I've fixed it
% sorting both sets before processing if they are bound variables.
% ------------------------------------------------------------------------------

powerset_subset( Subset, Set ) :-
  safe_sort( Subset, Sorted_Subset ),
  safe_sort( Set, Sorted_Set ),
  power_set( Sorted_Set, Powerset ), !,
  member( Sorted_Subset, Powerset ).

% ------------------------------------------------------------------------------
% bratko_subset( [ Subset ], [ Set ] )
% Backtrackable subset predicate based on Ivan Bratko's book "Prolog
% Programming for Artificial Intelligence". Its performance is slightly better
% than the powerset-based subset predicate.
%
% Note: it doesn't work if both sets are not in the same order; I've fixed it
% sorting both sets before processing if they are bound variables.
% ------------------------------------------------------------------------------

bratko_subset( X, Y ) :-
  safe_sort( X, Sorted_X ),
  safe_sort( Y, Sorted_Y ),
  actual_bratko_subset( Sorted_X, Sorted_Y ).

actual_bratko_subset( [], [] ).

actual_bratko_subset( [ First | Sub ], [ First | Rest ] ) :-
  actual_bratko_subset( Sub, Rest ).

actual_bratko_subset( Sub, [ _ | Rest ] ) :-
  actual_bratko_subset( Sub, Rest ).
```

```
% ------------------------------------------------------------------------------
% safe_sort( [ List ], [ List ] )
% It sorts a list if it is bound.
% ------------------------------------------------------------------------------

safe_sort( X, Sorted_X ) :-
  var( X ), !,
  Sorted_X = X.

safe_sort( X, Sorted_X ) :-
  sort( X, Sorted_X ).

% ------------------------------------------------------------------------------
% for_each_sort( [ [ Set ] ], [ [ Set ] ] )
% It internally sorts the sets in the first superset into the second superset.
% ------------------------------------------------------------------------------

for_each_sort( [], [] ).

for_each_sort( [ Xi | X ], [ Yi | Y ] ) :-
  sort( Xi, Yi ),
  for_each_sort( X, Y ).

% ------------------------------------------------------------------------------
% for_each_union( Goal, [ Set ], [ Set ] )
% It applies the goal of the form Goal( In, Out ) to all the members of the
% first set and stores the results in the second set.
% ------------------------------------------------------------------------------

for_each_union( _, [], [] ).

for_each_union( G, [ Xi | X ], Y ) :-
  call( G, Xi, Y1 ),
  for_each_union( G, X, Y2 ),
  union( Y1, Y2, Y ).

% ------------------------------------------------------------------------------
% for_each_sum( Goal, [ Set ], N )
% for_each_sum( Goal, K, [ Set ], N )
% It applies the goal of the form Goal( In, Out ) or Goal( K, In, Out ) to all
% the members of the set and sums the partial results into N.
% ------------------------------------------------------------------------------

for_each_sum( _, [], 0 ).

for_each_sum( G, [ Xi | X ], N ) :-
  call( G, Xi, N1 ),
  for_each_sum( G, X, N2 ),
  N is N1 + N2.

for_each_sum( _, _, [], 0 ).

for_each_sum( G, K, [ Xi | X ], N ) :-
  call( G, K, Xi, N1 ),
  for_each_sum( G, K, X, N2 ),
  N is N1 + N2.
```