



PReCISE – FUNDP  
University of Namur  
Rue Grandgagnage, 21  
B-5000 Namur  
Belgium

## TECHNICAL REPORT

January 20, 2010

AUTHORS	A. Classen, Q. Boucher, P. Faber, P. Heymans
APPROVED BY	P. Heymans
EMAILS	{acs, qbo, pfaber, phe}@info.fundp.ac.be
STATUS	Extended version of a paper appearing in the proceedings of the <i>Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)</i> , Linz, Austria.
REFERENCE	P-CS-TR SPLBT-00000003
PROJECT	MoVES
FUNDING	FNRS, the Walloon Region, Interuniversity Attraction Poles Programme of the Belgian State of Belgian Science Policy

### **Syntax and Semantics of TVL, a Text-based Feature Modelling Language**

# Syntax and Semantics of TVL, a Text-based Feature Modelling Language\*

Andreas Classen,<sup>†</sup> Quentin Boucher, Paul Faber and Patrick Heymans

PRECISe Research Centre  
Faculty of Computer Science  
University of Namur  
5000 Namur, Belgium

Email: {acs,qbo,pfaber,phe}@info.fundp.ac.be

**Abstract**—Feature models are a common way to represent variability in software product line engineering. For this purpose, most authors use a graphical notation based on FODA. The main drawback of those approaches is their lack of scalability: they generally do not fit real-size problems. Indeed, their graphical syntax does not account for attributes or complex constraints and becomes a burden for large feature models.

In this paper, we present TVL, a text-based feature modelling notation that is both light and comprehensive, meaning that it covers most constructs of existing languages, including cardinality-based decomposition and feature attributes. The main objective of TVL is to provide engineers with a human-readable language supporting large-scale models through modularisation mechanisms. Furthermore, TVL can serve as an extensible storage format for feature modelling tools.

We illustrate the various concepts of the language with short code fragments, provide a full EBNF grammar as well as a formal abstract syntax and semantics.

## I. INTRODUCTION

In software product line engineering (SPLE), Feature Models (FMs) are a common means to represent the variability of a software product line (SPL) [2]. Almost all existing FM languages are graphical notations based on FODA Feature Diagrams (FDs) which were introduced in the seminal paper by Kang *et al.* [3]. Since this original proposal, several extensions have been proposed by various authors. In all those dialects, FMs are represented as trees whose nodes denote features and whose edges represent top-down hierarchical decomposition of features. Consider the example FD in Figure 1 modelling a product line of personal computers. The *Computer* consists of a *Motherboard*, a *CPU*, a *Graphic Card* and some *Accessories*, which are optional (indicated by the hollow circle); all of these features are then further decomposed. In addition, although not shown in the figure, each of the features has a *price*, which can be modelled as an attribute [4].

While such a graphical representation is supposedly more accessible to non-technical stakeholders, we believe that working with large industry-size FDs can become a tricky task for several reasons. First, to create a large FD, the graphical syntax is a burden that cannot be mastered without dedicated

tool support (though many FM tools use directory tree-like representations themselves). Secondly, given that a FD is a tree on a two-dimensional surface, there will inevitably be large physical distances between features, which makes it hard to navigate, search or interpret the FD. Finally, most notations do not have graphical means to represent constructs like attributes and constraints which are essential for industrial FMs.

Better user interfaces or visualisation techniques have been proposed as a remedy for many of these problems [5], but they have their drawbacks too. They rely on software to visualise a model, meaning that without this software, as for instance on a blackboard, on paper, on a random computer, they will not work. Furthermore, software-intensive modelling solutions (e.g., commercial CASE tools or existing FM tools) tend to offer poor interoperability which can be an obstacle to collaborative work between engineers, due in part to opaque file formats that prevent efficient information exchange and meaningful versioning.

Here, instead of attacking the symptoms of the problem, we attack its cause and propose to use another FM representation. We do not question the need for graphical representations. We rather propose a textual notation, supporting feature attributes, which could be used in combination with graphical ones. This would allow one to view and edit FMs either graphically or textually, depending on one's skills and preferences. We thereby hope to facilitate the dissemination of FMs in industrial settings. The goals for the new language are to be *scalable* by being *succinct* and *modular* as well as *comprehensive*.

The fundamental question being the medium of the language, we decided to pick a medium that is well-accepted and for which a very large number of applications (with no interoperability problems) exist: plain text. Our language is called TVL, for *text-based variability language*. Plain text has a number of advantages the most important of which is the abundance of established tools dealing with text, generally program code. Moreover, the syntax of TVL is inspired by the syntax of C and should appear intuitive to any engineer who has come in contact with one of the many programming languages with a C-like syntax. We believe that these choices will also ease acceptance of FMs in industrial contexts because TVL is similar to the languages used in such environments

\* A short version of this paper appears in the proceedings of the *Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*, Linz, Austria, January 2010 [1].

<sup>†</sup> FNRS Research Fellow

and because it does not need dedicated modelling tools to be deployed.

The TVL language is *scalable* because it is *succinct* (its syntax is very light, as opposed to XML, for instance) and because it offers a number of mechanisms for *modularisation* and separation of concerns: it allows to spread an FM over several files, to break down the hierarchy of an FM into several smaller hierarchies and to spread the definition of a feature over several instances. The language is *comprehensive* because it integrates most of the FM constructs proposed in the twenty years since the advent of FODA.

At this stage, TVL is a language proposal. It is formally defined with an LALR grammar, a formal semantics and comes with a reference implementation available online.<sup>1</sup> It is meant to be a basis for discussion and we are mainly interested in feedback about the language and its syntax.

The remainder of the paper is structured as follows: Section II introduces the TVL syntax with example code snippets before the formal EBNF grammar is specified in Section III. Section IV gives well-formedness rules for TVL models. A formal semantics for TVL follows in Section V, related work is surveyed in Section VI and the paper concluded in Section VII.

## II. SYNTAX

In this section we present an overview of the TVL syntax using code snippets before giving the formal BNF grammar. The following sub-sections introduce five major parts of the language: features, attributes, expressions, constraints and modularisation mechanisms.

The different concepts of TVL will be illustrated using a basic personal computer product family example FD presented in Section I and visible in Figure 1.

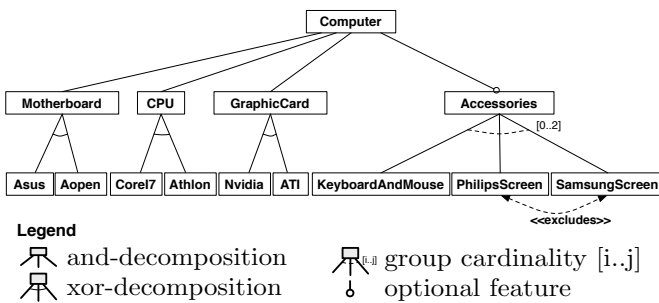


Fig. 1. Computer example FD

### A. Feature hierarchy

The TVL language has a C-like syntax: it uses braces to delimit blocks, C-style comments, semicolons to delimit statements. The rationale for this syntax choice is that nearly all computing professionals have come across a C-like syntax and are thus familiar with this style. Furthermore, many text editors have built-in facilities to handle this type of syntax.

In our example, the root feature, *Computer*, is decomposed into four sub-features by an *and*-decomposition: *Motherboard*, *CPU*, *GraphicCard* and *Accessories*. Furthermore, the *Accessories* feature is optional while the other three features are mandatory. In TVL, this is written as follows:

```

root Computer {
  group allOf {
    Motherboard,
    CPU,
    GraphicCard,
    opt Accessories
  }
}

```

A decomposition type in TVL is defined with the **group** keyword. Predefined decomposition operators are **allOf**, as used in this example for an *and*-decomposition, **oneOf** for *xor*-decompositions and **someOf** for *or*-decompositions. It is also possible to specify a cardinality-based decomposition with the **group [i..j]** syntax, where *i* and *j* are the lower and upper bounds of the cardinality. When defining a cardinality, one can use the asterisk character \* to denote the number of children in the group, for instance **group [1..\*]** would be equivalent to **group someOf**. This way, the engineer does not have to update the cardinality each time the number of children changes. Optional features like *Accessories* are declared by putting the **opt** keyword in front of their name.

FMs most commonly have a tree structure but, sometimes, a directed acyclic graph (DAG) structure – a feature can have several parents – might be useful [6]. DAG structures can also be modelled in TVL with the **shared** keyword associated to a feature name. This means that the shared feature has several parents as it is illustrated in the following example where feature *D* has features *B* and *C* as parents:

```

root A
  group oneOf {
    B group allOf {D},
    C group allOf {shared D}
  }
}

```

### B. Attributes

In our example, the *Motherboard* has four attributes: a price, a width, an height and a socket type. TVL supports four different attribute types: integer (**int**), real (**real**), Boolean (**bool**) and enumeration (**enum**). In our example, *price*, *width* and *height* are integers. Furthermore, the *price* value is limited to values between 0 and 500. In TVL, this is expressed as follows:

```

Motherboard {
  int price in [0..500];
  int width;
  int height;
}

```

Attributes are thus declared by defining their type and name inside the definition block of the feature they belong to. Each attribute declaration is terminated by a semicolon. The **in** keyword is optional, it can be used to restrict the domain of an attribute (which might speed up automated reasoning). When declaring an attribute as an enumeration type, this means that it

<sup>1</sup>Download at the TVL website <http://www.info.fundp.ac.be/~acs/tvl>.

will take exactly one of a set of predefined values. The *socket*, for instance, is either *LGA1156* or *ASB1*. We thus declare it as an enum.

```
Motherboard {
    enum socket in {LGA1156, ASB1};
}
```

For enumerations, the **in** keyword is mandatory. Notice the use of curly braces here as opposed to square brackets for the *price* attribute above. In TVL, square brackets are used to declare intervals and braces to declare lists. Enumeration attributes are very similar to *xor*-decomposed features: they can be seen as a shorthand notation which avoids clutter and boilerplate code.

In many cases, the value of an attribute will be calculated based on the values of some other attributes. The value of the *price* attribute of *Accessories*, for example, is the sum of the prices of its children *KeyboardAndMouse*, *PhilipsScreen* and *SamsungScreen*. Furthermore, the value of an attribute might also depend on whether its containing feature is selected or not. All this is written as follows in TVL:

```
Accessories {
    int price is sum(selectedChildren.price);
    group [0..2] {
        KeyboardAndMouse {
            int price is 19;
        },
        PhilipsScreen {
            int price is 99;
        },
        SamsungScreen {
            int price, ifIn: is 149, ifOut: is 0;
        }
    }
}
```

The keyword **is** can be used to set the value of an attribute, e.g. *Accessories*, *KeyboardAndMouse* and *SamsungScreen*. The keywords **ifIn:** and **ifOut:** are guards that allow to specify the value of the attribute in the case in which the containing feature is selected (**ifIn:**) or not selected (**ifOut:**). We illustrate this with the *price* attribute of the *SamsungScreen* whose value will be 149 if the feature is selected and 0 if not.

While the price of the *KeyboardAndMouse*, *PhilipsScreen* and *SamsungScreen* features is fixed, the price of the *Accessories* is calculated: it is the sum (using the aggregation function **sum**) of the values of the *price* attribute of its selected children (using the **selectedChildren** keyword, which basically represents the list of values of an attribute declared in all the selected child features). Other operators are available and will be discussed in next section. A common modelling pattern for attributes declared for all features is to compute the value of the parent feature’s attribute by aggregating the attribute values of its children, up to the root. The price of a *Computer*, for example, will be calculated by summing the prices of its selected sub-features, which in turn depend on the prices of their sub-features, and so on until leaf features with fixed price values are reached.

### C. Expressions

In TVL, expressions are used to determine the value of an attribute (as explained in the previous section) as well as to express constraints on the FM (detailed in the following section). The language is strongly typed, each expression being either of type *bool*, *real* or *int*. For more info about types see Section IV.

A basic expression is either an integer, a real, a Boolean, or a reference to a feature, an attribute or a constant. Those basic expressions can then be combined using classical operators: **+**, **-**, **/**, **\***, **abs**, for numeric values; **!**, **&&**, **||**, **->**, **<->** for Boolean values as well as comparison operators **>**, **>=**, **<** or **<=**. Classical FM cross-tree constraints **excludes** and **requires** can also be used as Boolean expressions.

Furthermore, there are a number of aggregation functions **sum**, **mul** (multiplication), **min**, **max**, **avg** (average), **count**, **and**, **or** and **xor**. These aggregation functions can simply be used on lists of expressions or they can become powerful shorthand notations when used in combination with the **children** or the **selectedChildren** keywords. These allow to aggregate the value of an attribute that is declared for each child of a feature. The notation is **fct(children.attribute)**, or **fct(selectedChildren.attribute)** if the aggregate should be calculated on selected children only.

A full listing of the expression syntax is given in Section III-E.

### D. Constraints

Constraints in TVL are attached to features (classical constraints that hold ‘for the whole model’ can be attached to the root, for instance). They are simply Boolean expressions that can be added to the body of a feature definition, as with attribute declarations. They are terminated by a semicolon. The **ifIn:** and **ifOut:** guards we have previously seen can be used on constraints, too. In our computer example, the *Motherboard* feature has a *socket* attribute. The value of this attribute depends on the choice of the actual motherboard, i.e. on the choice of one of the sub-features. One way to model this in TVL is to define a constraint in each child feature which basically ‘sets’ the value of its parent’s attribute.

```
Motherboard {
    enum socket in {LGA1156, ASB1};
    group oneOf {
        Asus {
            ifIn: parent.socket == LGA1156;
        },
        Aopen {
            ifIn: parent.socket == ASB1;
        }
    }
}
```

The constraint is guarded by **ifIn:**, which means that it is only applicable if the containing feature is selected.

### E. Modularisation mechanisms

One of the main goals in designing TVL is modularity (to achieve scalability). TVL thus offers various mechanisms that can help users to modularise large models. First of all, custom

types can be defined on at the top of the file and then be used in the FM. This allows to factor out recurring types and can thus reduce consistency errors. For instance, one might want to define the different sockets upfront and then use it as a type in an attribute declaration:

```
enum cpuSocket {LGA1156, ASB1};
...
Motherboard {
    cpuSocket socket;
}
```

It is possible to define structured types to group attributes that are logically linked. A *dimension*, for instance, is a couple (height, width) and can be declared as such using a structured type. This type can then be reused inside the *Motherboard* feature:

```
struct dimension {
    int height;
    int width;
}
...
Motherboard {
    dimension size;
}
```

Users can also specify constants using the **const** keyword followed by a type, a name and a value. These constants can then be used inside expressions or cardinalities.

```
const int maxRamBlocks 4;
```

Modularisation is achieved through two distinct mechanisms. The first is the **include** statement, which takes as parameter the path of a file (relative to the file containing the root feature). As expected, an **include** statement will include the contents of the referenced file at this point. Includes are in fact preprocessing directives and do not have any meaning beyond the fact that they are replaced by the referenced file.

```
include(./some/other/file);
```

The second mechanism is that features can be defined at one place and then extended further in the code. This has two consequences: the definition of a feature can be spread over a number of blocks and the physical hierarchy of the FM does not have to be maintained inside the code (for instance, to break up deeply nested hierarchies requiring lots of indentations).

Basically, once a feature has been defined in the group block of its parent feature, its definition can be extended any number of times. In order to extend a feature definition, one just adds a feature block with the same name to the file. This block cannot be inside another feature, it has to start its own hierarchy. Each feature block may add constraints and attributes to the feature body. The children (with the **group** keyword) can only be defined in a single one of these blocks.

This mechanism allows modellers to organise the FM according to their preferences and can be used to implement separation of concerns [7]. For example, one could separate different attribute concerns (e.g. attributes related to price and attributes related to technical details, like the sockets). Another scenario would be to declare part of the structure of the FM

without detailing each feature's attributes and instead provide them later on:

```
root Computer {
    group allOf {
        Motherboard,
        CPU,
        GraphicCard,
        opt Accessories
    }
}
Computer {
    int price is sum(selectedChildren.price);
}
...
Motherboard {
    dimension size;
    ...
}
...
```

In this example, the decomposition of top feature *Computer* is defined at the beginning while its attributes and those of *Motherboard* are declared further down. The advantage of this is that the structure is easily understandable because it is not cluttered by the attributes of the different features.

### III. GRAMMAR

The grammar is a conflict-free LALR grammar. To resolve some of the conflicts, it needs operator priorities which are given in Table II, Section V. These priorities are not further discussed here since they are rather standard and not of interest to the discussion.

The grammar is given in extended Backus-Naur form (EBNF): terminals are enclosed in double quotes, parentheses are used for grouping, [S] means S is optional, (S)+ means that S repeats one or more times and (S)\* is a shortcut for [(S)\*]. To make the rules more readable, non-terminals are written in uppercase.

The starting non-terminal is the model, and quite naturally, a model is a sequence of type, constant and feature declarations.

```
MODEL = ( TYPE | CONSTANT | FEATURE )*
```

#### A. Type and constant declarations

A type is either a simple type (i.e. it simply renames a basic type), or a structured type with several fields. The simple types have an ID and can have a domain (declared with the **in** terminal). A structured type is just a list of simple types in curly braces, with the exception that a structured type can make use of already declared simple types (the RECORD\_FIELD = ID ID ";" production below).

```
TYPE = SIMPLE_TYPE
    | RECORD ;

SIMPLE_TYPE = "int" ID "in" SET_EXPRESSION ";"
    | "real" ID "in" SET_EXPRESSION ";"
    | "enum" ID "in" SET_EXPRESSION ";"
    | "int" ID ";"
    | "real" ID ";"
    | "bool" ID ";" ;

RECORD = "struct" ID "{" RECORD_FIELD+ "}" ;

RECORD_FIELD = SIMPLE_TYPE
    | ID ID ";" ;
```

As expected, constants consist of the **const** terminal, their type, their identifier and their value.

```
CONSTANT = "const" "int" ID INTEGER ";"
          | "const" "real" ID REAL ";"
          | "const" "bool" ID ( "true" | "false" ) ";"
```

## B. Identifiers

The non-terminal ID can refer to types, constants, but also to features and feature attributes. IDs have to start with a character and can contain numbers as well as the underscore. There are two rules that are not formalised in the grammar and will be detailed in Section IV: feature IDs have to start with an uppercase letter and IDs in TVL are case sensitive.

```
ID = ("a"-"z" | "A"-"Z" |
      "a"-"z" | "A"-"Z" | "0-9" | "_" ) * ;
```

In order to allow feature names to occur several times, and to allow references to an attribute of a feature inside the body of another feature, it is possible to build chains of IDs separated with a dot. All IDs of a chain with at least two IDs, except for the last one, have to denote features. The last element may be a feature or an attribute. In this context, the terminals **root**, **this** and **parent** help to make specifications more intuitive, as seen in Section II-D.

```
SHORT_ID = "root"
          | "this"
          | "parent"
          | ID ;
```

```
LONG_ID = SHORT_ID
          | SHORT_ID "." LONG_ID ;
```

## C. Feature declarations

A feature declaration consists of an ID (optionally preceded by the **root** terminal) which is followed either by its body (that is, a number of `FEATURE_BODY_ITEMS`), or directly by its children block (the `FEATURE_GROUP` non-terminal) if one wants to omit attribute or constant declarations. In case a feature is extended rather than declared for the first time, it may use a long ID since the feature being extended might not have a unique name. When extending the root this way, the **root** terminal must be omitted.

```
FEATURE = "root" ID "{" FEATURE_BODY_ITEM+ "}"
          | "root" ID FEATURE_GROUP
          | LONG_ID "{" FEATURE_BODY_ITEM+ "}"
          | LONG_ID FEATURE_GROUP ;
```

The feature body consists of several items which can be data blocks, constraints, attributes or the group block declaring the child features.

```
FEATURE_BODY_ITEM = DATA
                  | CONSTRAINT
                  | ATTRIBUTE
                  | FEATURE_GROUP ;
```

The children body starts with the **group** terminal, followed by the cardinality and the list of children. A cardinality can be either a predefined one, or an interval of natural numbers which can be specified directly, with natural numbers, constants (the `ID` non-terminal) or the asterisk character as explained in Section II-A.

```
FEATURE_GROUP = "group" CARDINALITY "{"
               HIERARCHICAL_FEATURE
               ( "," HIERARCHICAL_FEATURE ) *
               "}" ;

HIERARCHICAL_FEATURE = ( "opt" ) ? FEATURE
                       | ( "shared" | "opt" ) ? LONG_ID ;

CARDINALITY = "oneof"
             | "someof"
             | "allof"
             | "[" ( ID | NATURAL | "*" ) ".."
                 ( ID | NATURAL | "*" ) "]" ;
```

## D. Attributes and constraints

An attribute declared for a feature is either a single attribute (`BASE_ATTRIBUTE`) or a structured attribute. A structured attribute has to be of a structured type defined previously (the first ID), have a name (the second ID) and may list the fields of that structured type to define a body for them. A single attribute is just a type followed by a name and, optionally, the body.

```
ATTRIBUTE = BASE_ATTRIBUTE
           | ID ID "{" SUB_ATTRIBUTE+ "}" ;

BASE_ATTRIBUTE = "int" ID ATTRIBUTE_BODY ? ";"
                | "real" ID ATTRIBUTE_BODY ? ";"
                | "bool" ID ATTRIBUTE_BODY ? ";"
                | "enum" ID ATTRIBUTE_BODY ? ";"
                | ID ID ATTRIBUTE_BODY ? ";" ;

SUB_ATTRIBUTE = ID ATTRIBUTE_BODY ";" ;
```

The attribute body allows to restrict the domain of an attribute, or to give it a value as part of the attribute declaration (instead of doing it in the constraints). A fixed value is defined with the **is** terminal. A domain is specified with the **in** terminal followed by a `SET_EXPRESSION` (second and third production). In this case, one can further specify a conditional value assignment (which does not make sense in the case the attribute value is fixed). Such a conditional value assignment can also be specified alone, i.e. without being preceded by an **is** or **in** statement.

```
ATTRIBUTE_BODY = "is" EXPRESSION
                | "in" SET_EXPRESSION
                ( "," ATTRIBUTE_CONDITIONAL ) ?
                | "," ATTRIBUTE_CONDITIONAL ;
```

A conditional value assignment allows to specify an **is** or an **in** statement that should hold depending on whether the feature in which the attribute is declared is selected (**ifin**: terminal) or not (**ifout**: terminal). If both cases are given, they have to be separated by a comma and should start with the **ifin**: terminal. The productions simply capture the eight possible combinations for this.

```
ATTRIBUTE_CONDITIONAL =
  "ifin:" "is" EXPRESSION
  "," "ifout:" "is" EXPRESSION
| "ifin:" "is" EXPRESSION
| "ifout:" "is" EXPRESSION
| "ifin:" "in" SET_EXPRESSION
  "," "ifout:" "is" EXPRESSION
| "ifin:" "is" EXPRESSION
  "," "ifout:" "in" SET_EXPRESSION
```

```

| "ifin:" "in" SET_EXPRESSION
  ", " ifout:" "in" SET_EXPRESSION
| "ifin:" "in" SET_EXPRESSION
| "ifout:" "in" SET_EXPRESSION ;

```

A constraint declaration is just a boolean expression terminated by a semicolon. Just as an attribute value declaration, it can be guarded with the **ifin:** or **ifout:** terminals.

```

CONSTRAINT = EXPRESSION ";"
             | "ifin:" EXPRESSION ";"
             | "ifout:" EXPRESSION ";" ;

```

## E. Expressions

The expression syntax is meant to be as complete as possible in terms of operators, to encourage writing of intuitive constraints. The meaning of most productions should be clear.

```

EXPRESSION =
  (* Reference to a feature/an attribute *)
  LONG_ID

  (* Grouping *)
  | "(" EXPRESSION ")"

  (* Classical FM constraints *)
  | LONG_ID "excludes" LONG_ID
  | LONG_ID "requires" LONG_ID

  (* Conditional expression *)
  | EXPRESSION "?" EXPRESSION ":" EXPRESSION

  (* Boolean *)
  | EXPRESSION "&&" EXPRESSION
  | EXPRESSION "||" EXPRESSION
  | EXPRESSION "->" EXPRESSION (* implication *)
  | EXPRESSION "<-" EXPRESSION
  | EXPRESSION "<->" EXPRESSION (* equivalence *)
  | "!" EXPRESSION
  | "true"
  | "false"

  (* Boolean aggregation *)
  | "and" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
  | "or" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
  | "xor" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"

  (* Comparison *)
  | EXPRESSION "==" EXPRESSION
  | EXPRESSION "!=" EXPRESSION
  | EXPRESSION "<=" EXPRESSION
  | EXPRESSION "<" EXPRESSION
  | EXPRESSION ">=" EXPRESSION
  | EXPRESSION ">" EXPRESSION

  (* Domain restriction *)
  | EXPRESSION "in" SET_EXPRESSION

  (* Arithmetic *)
  | EXPRESSION "+" EXPRESSION
  | EXPRESSION "-" EXPRESSION
  | EXPRESSION "/" EXPRESSION
  | EXPRESSION "*" EXPRESSION
  | "-" EXPRESSION
  | "abs" "(" EXPRESSION ")"
  | INTEGER
  | REAL

  (* Arithmetic aggregation *)
  | "sum" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
  | "mul" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
  | "min" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"

```

```

| "max" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
| "count" "(" ("children" | "selectedchildren") ")"
| "avg" "(" (EXPRESSION_LIST | CHILDREN_ID) ")" ;

```

The **in** statements that can be used to define or restrict the domain of an attribute require a `SET_EXPRESSION` that specifies the set of values. A set expression can either be a list of expressions inside curly braces (i.e. the set is defined *in extension*) or an interval with integer, real or infinite (the `*` character) bounds between square brackets (to define a set *in intension*).

```

SET_EXPRESSION =
  "{" EXPRESSION_LIST "}"
  | "[" (INTEGER | REAL | "*" ) ".."
      (INTEGER | REAL | "*" ) "]" ;

```

An expression list is just a comma-separated list of expressions. Its main use is in defining sets in extension, but it can also be used in combination with an aggregation function. The domain of an enum (i.e. the values of an enum) is actually an expression list where every expression is an ID non-terminal.

```

EXPRESSION_LIST = EXPRESSION ("," EXPRESSION_LIST)* ;

```

To concisely specify cases in which the value of an attribute is an aggregate of another attribute that is declared for each child, the **children** statement can be used (followed by a `LONG_ID` denoting the attribute).

```

CHILDREN_ID = "selectedchildren" "." LONG_ID
             | "children" "." LONG_ID ;

```

## F. Data blocks

Data blocks were not further detailed in the previous section. They are part of the language but do not have a meaning in terms of FMs. Their purpose is rather to allow TVL to be an extensible file format for FMs. They make it possible to attach to every feature a catalogue of *key/value* pairs that can be used to store additional, tool-specific, data. If, for instance, TVL were to be used as the storage format for a graphical FM tool, the data block of a feature might contain the coordinates of the feature on the screen and other style information.

A data block starts appropriately with the **data** terminal, followed by a key/value list in curly braces where key and value are separated by a blank and each pair is terminated by a semicolon. There can be several data blocks in each feature, all being merged when the diagram is parsed. Keys should be unique for each feature. As noted before, data keys do not have any meaning in the normal FD semantics, they cannot be ‘referenced’ anywhere in the model.

```

DATA = "data" "{" DATA_PAIR+ "}" ;
DATA_PAIR = STRING STRING ";" ;

```

## G. Values

For completeness sake, we also give the rules for naturals, integers, reals and strings which are rather standard. Strings have to be enclosed in double quotes; double quotes can be escaped with a backslash.

```

NATURAL = "0" | ["1"-"9"] ["0"-"9"]* ;
INTEGER = "0" | ("-"?) ["1"-"9"] ["0"-"9"]* ;
REAL    = INTEGER "." (["0"-"9"]* ["1"-"9"])? ;
STRING  = "'" [^]' ' ' ;

```

#### IV. WELL-FORMEDNESS

There are a number of rules to which a TVL model has to adhere in order to be valid.

##### A. Naming, scope and references

The naming rules for names of features, attributes, types, constants, enum values and struct fields are similar to other C-like languages: they can use letters, digits, the underscore and cannot begin with a digit; names are case-sensitive. Furthermore, there is a list of reserved keywords that may not be used.

There is one mandatory naming convention, namely that feature names must begin with a capital letter while names of attributes, types, constants and struct fields must begin with a lowercase letter (enum values are exempt from this rule). This prevents a number of potential naming conflicts. Of course, names have to be unique within their scope, that is,

- child features and attributes of the same feature,
- all declared types,
- constants,
- struct fields that are siblings,
- enum values per enum type.

are all required to have distinct names. Furthermore,

- no enum value may have the name of an attribute, a feature or a constant
- no constant may have the name of an attribute.

Feature and attribute names, however, do not have to be globally unique except for the name of the root feature. To reference a feature called *bar* inside a constraint, one can just use its name, i.e. write *bar*, or use a *qualified name*. Assuming its parent feature to be called *foo*, a qualified name would be *foo.bar*, or *baz.foo.bar* if *baz* is the parent of *foo*; and so on. A *fully qualified name* is one that starts with the name of the root feature.

Each reference to a feature inside a constraint has to be unambiguous. This means that if the name of the referenced feature is unique, it is sufficient to put this name. Otherwise, an unambiguous qualified name has to be used, that is, the feature name has to be prefixed by the names of its parents until the uppermost name is unique. Since the name of the root feature has to be unique, a fully qualified name is always unambiguous.

The rules for the referencing of attributes are similar. Inside the body of their containing feature they can be referenced solely with their name. Otherwise they have to be prefixed by the name of their containing feature. If this name is ambiguous, the same rules as above apply.

There are a number of keywords to make referencing easier: **parent** denotes the parent feature of the feature in the body of which it is used, **root** always denotes the root feature and **this** denotes the feature in the body of which it is used.

##### B. Type correctness

TVL is strongly typed, and does not allow type casting, furthermore, TVL type checking can be performed statically. Type correctness is defined as expected: expressions defining the value of an attribute have to be of the same type as the attribute. Constraints have to be expressions of type *bool*. The expressions themselves have to be correctly typed, that is, Boolean operators may only take Boolean operands, numeric operators may only take numeric operands, and so on.

When a set is defined in extension, i.e. with a list of elements, all elements in the list need to have the same type, which is the type of the set. Expressions involving attributes of type *enum* may only use enum values defined for the attribute.

##### C. Other rules

The decomposition relation of the model has to be acyclic. While the grammar allows to declare a cycle in the decomposition relation, this is not permitted. Decomposition cardinalities  $\langle i, j \rangle$  have to be so that  $i \leq j$  and  $i$  has to be smaller than or equal to the number of child features. There can only be one **group** block per feature. The **parent** keyword may not be used for features having more than one parent.

The **children** keyword can be used in combination with an aggregation function to apply the function to the value of the attribute of all children of the feature. Its use therefore requires that the attribute is indeed declared for all the children of the feature, that the children all declare it with the same type and that this type is compatible with the aggregation function. The **selectedChildren** keyword is similar, except for the fact that it ranges only over children that were selected. The same rules apply here. In addition, this keyword cannot be used for the **min** and **max** aggregation functions and it can only be used if the decomposition relation of the parent feature is so that there will always be at least one selected child.

In the following section, we consider that models are compliant with all those well-formedness rules.

## V. SEMANTICS

In line with previous work of the authors [8], [9], a language is not fully defined without a formal semantics. Fortunately, most of the work has already been done elsewhere, mainly by Schobbens *et al.* [8] with the formal definition of Free Feature Diagrams (FFD), a parametrised language that was used to give a formal semantics to a wide range of FD languages in one shot.

We cannot reuse the FFD definition as is. First of all, FFD are based on an abstract syntax that is much more limited than the concrete syntax of TVL. In Section V-A, we thus define a translation from TVL to an abstract syntax close to that of FFD. Furthermore, FFD lack a number of concepts that were introduced by various authors over time and integrated into the TVL language. For instance, they do not formalise attributes or non-Boolean constraints. Also, they do not explicitly capture the notion of *optional* feature, which they encode with an intermediate dummy feature that has a [0-1] decomposition

to a unique child. We contribute these missing pieces in Section V-B.

For the definition of the semantics, we follow the guidelines of Harel and Rumpe [10], meaning that we formally define the abstract syntax  $\mathcal{L}$  of our language, the semantic domain  $\mathcal{S}$  and the semantic function  $\mathcal{M} : \mathcal{L} \rightarrow \mathcal{S}$ .

#### A. Abstract syntax $\mathcal{L}_{TVL}$

The concrete syntax introduced in the previous section offers a number of modularisation mechanisms, types, constants, guarded constraints as well as other kinds of syntactic shortcuts. In order to obtain an easily formalisable language, the abstract syntax for TVL will abstract away from these, resulting in a “purer” language with less constructs.

**Definition 1** (TVL Abstract Syntax, extension of [8]). *The syntactical domain  $\mathcal{L}_{TVL}$  is the set of all tuples  $(N, r, DE, \omega, \lambda, A, \rho, \tau, \Phi)$  where:*

- $N$  is the (non empty) set of features,
- $r \in N$  is the root,
- $DE \subseteq N \times N$  is the decomposition (hierarchy) relation between features. It can be a DAG. For convenience, we will write  $n \rightarrow n'$  sometimes instead of  $(n, n') \in DE$ ,
- $\omega : N \rightarrow \{0, 1\}$  labels optional features with a 1,
- $\lambda : N \rightarrow \mathbb{N} \times \mathbb{N}$  indicates the decomposition operator of a feature, represented as a cardinality  $\langle i..j \rangle$  where  $i$  indicates the minimum number of children required in a configuration and  $j$  the maximum (we use angle brackets to distinguish cardinalities from other tuples),
- $A$  is the set of attributes,
- $\rho : A \rightarrow N$  is a total function that returns the feature to which an attribute is attached,
- $\tau : A \rightarrow \{int, real, enum, bool\}$  is a total function giving the type of each attribute,
- $\Phi \in \mathcal{L}_{exp}$  is a Boolean expression over the features  $N$  and the attributes  $A$ , expressing additional constraints on the model. Details are provided in Definition 2.

Furthermore, each  $d$  must satisfy the following well-formedness rules:

- $r$  is the unique root  $\forall n \in N (\nexists n' \in N \bullet n \rightarrow n') \Leftrightarrow n = r$ ,
- $r$  is not optional  $\omega(r) = 0$ ,
- $DE$  is acyclic  $\nexists n_1, \dots, n_k \in N \bullet n_1 \rightarrow \dots \rightarrow n_k \rightarrow n_1$ ,
- Terminal nodes are  $\langle 0..0 \rangle$ -decomposed.

There is one element of the syntax that we have somewhat neglected. In fact, we define the additional constraint  $\Phi$  as a Boolean expression over the features and attributes. The TVL language, however, has very precisely defined syntax for  $\Phi$  for which we have to provide an abstract syntax and semantics.

**Definition 2** (Syntax and semantics of expressions).  $\mathcal{L}_{exp}$  is the set of all correctly typed Boolean expressions  $B$  over the set  $N$  of features and  $A$  of attributes that are formed according to the grammar given in Table I, where  $n \in N$  is a feature,  $a \in A$  is an attribute,  $d \in \mathbb{Z}$  is an integer,  $q \in \mathbb{Q}$  is a rational number and  $t$  is an enum value. Since the expression language

TABLE I  
EXPRESSION SYNTAX OF  $\mathcal{L}_{TVL}$

$$\begin{aligned}
B ::= & \text{true} \mid \text{false} \mid n \mid a \mid t \mid E \text{ in } S \mid \\
& n \text{ excludes } n \mid n \text{ requires } n \mid \\
& B \ \&\& \ B \mid B \ \parallel \ B \mid ! \ B \mid \\
& B \ \rightarrow \ B \mid B \ \leftarrow \ B \mid B \ \leftrightarrow \ B \mid \\
& E \ == \ E \mid E \ != \ E \mid \\
& E \ < \ = \ E \mid E \ < \ E \mid E \ > \ = \ E \mid E \ > \ E \mid \\
& \text{and}(B \ [ \ , \ B]^* ) \mid \text{or}(B \ [ \ , \ B]^* ) \mid \text{xor}(B \ [ \ , \ B]^* ) \\
E ::= & n \mid a \mid t \mid d \mid q \mid \\
& E \ + \ E \mid E \ - \ E \mid E \ / \ E \mid E \ * \ E \mid - \ E \mid \\
& \text{abs}(E) \mid B \ ? \ E : \ E \mid \\
& \text{sum}(E \ [ \ , \ E]^* ) \mid \text{mul}(E \ [ \ , \ E]^* ) \mid \\
& \text{min}(E \ [ \ , \ E]^* ) \mid \text{max}(E \ [ \ , \ E]^* ) \\
S ::= & \{ E \ [ \ , \ E]^* \} \mid \\
& [ (d \mid *) \ .. (d \mid *) ] \mid [ (f \mid *) \ .. (f \mid *) ]
\end{aligned}$$

uses only standard operators, its semantics is straightforward and omitted here for the sake of brevity. Just keep in mind that TVL is purely declarative, and that each expression in TVL is in fact a Boolean expression.

We will define a formal semantics for  $\mathcal{L}_{TVL}$  and  $\mathcal{L}_{exp}$  only. The  $\mathcal{L}_{TVL}$  language (abstract syntax) is in fact a subset of the TVL language (concrete syntax) defined in the previous section. In the following, we will show that any TVL model can be transformed into an equivalent model using only the subset of constructs defined for  $\mathcal{L}_{TVL}$ . We thereby formally map the concrete syntax to the abstract syntax. We say that a TVL model using only constructs from  $\mathcal{L}_{TVL}$  is in *normal form*, and the subset of the TVL language reduced to models in normal form is called  $TVL_{NF}$ . The semantics of the TVL language is thus provided in two steps. A first step is to provide a formal semantics to the many constructs that are not part of the normal form  $TVL_{NF}$ . We will do this by providing a syntactic translation from TVL to  $TVL_{NF}$ . The second step is to define the semantics of  $TVL_{NF}$ , i.e. that of  $\mathcal{L}_{TVL}$ .

$TVL_{NF}$  has the same syntax as TVL. However, the only allowed constructs are those defining the features and their hierarchy ( $N$ ,  $r$  and  $DE$ ), optional features ( $\omega$ ), cardinality-based decomposition operators ( $\lambda$ ) and attributes with basic types ( $A$ ,  $\rho$  and  $\tau$ ). The excluded constructs are mainly all kinds of modularisation mechanisms and non-cardinality decomposition operators. Furthermore,  $TVL_{NF}$  has only one single constraint ( $\Phi$ ) that has to be an expression of  $\mathcal{L}_{exp}$ . To obtain an expression in  $\mathcal{L}_{exp}$  from a  $TVL_{NF}$  expression, we have to define operator precedence, associativity and parentheses, since Definition 2 abstracts away from these. We chose to define operator precedence in TVL and  $TVL_{NF}$  to be the same as in C.

**Definition 3** (Operator precedence in TVL and  $TVL_{NF}$ ). *Table II lists all operators in decreasing order of precedence. The associativity of each operator is given in the left column. Parentheses can be used to group expressions and force a different evaluation order.*

Now, the remaining constructs map 1:1 to the elements of  $\mathcal{L}_{TVL}$  and  $\mathcal{L}_{exp}$  in Definitions 1 and 2. The first part of the semantics is provided in Definition 4 which specifies how a

TABLE II  
OPERATOR PRECEDENCE IN TVL AND TVL<sub>NF</sub>

Associativity	Operators
right	!, (unary) -, and aggregation functions
non	requires, excludes
left	*, /
left	+, -
non	>, <, >=, <=
non	==, !=, in
left	&&
left	
non	<->
left	->
right	<-
right	? :

TVL model is translated into TVL<sub>NF</sub>. It thus describes how to translate constructs that only exist in TVL into those that exist in TVL<sub>NF</sub>, thereby defining their semantics.

**Definition 4.** A model in TVL<sub>NF</sub> is obtained from a model in TVL by applying the following transformation steps in the specified order.

- 1) **Includes.** Eliminate the **include** preprocessing directive by replacing it with the content of the referenced file.
- 2) **Constants.** Eliminate constants **const t c e**; by replacing all occurrences of **c** by its definition **e**.
- 3) **Types.** Here we distinguish between types that merely rename basic types **b t**; and more complex structured types **struct t {b1 t1, b2 t2,..}**. The former can be eliminated by replacing all occurrences of the defined type **t** by the corresponding basic type **b**. Structured types can be eliminated in a two step process. The first step is to flatten a structured type **t** by replacing it by a number of individual types **b1 t1; b2 t2;...**, and to flatten attributes declared as structs by replacing them by individual attributes. The flattened types are then eliminated in a recursive step.
- 4) **Attribute domain and value specifications.** The construct **t a in s**; allows to specify the range of an attribute **a** to be the set **s**. The **in** construct is removed and a constraint of the form **this.a in s**; is added. Similarly, the construct **t a is v**; allows to specify a fixed-value attribute **a** to be **v**. The **is** construct is removed and a constraint of the form **this.a == v**; is added.
- 5) **Conditional domain and value specifications.** An attribute value or domain specification can also be guarded with the keywords **ifIn:** and **ifOut:**, the syntax then is **t a, ifIn: vin, ifOut: vout**; where **vin** can be **in s** to specify a domain or **is v** to specify a value. These constructs are removed and constraints of the form **ifIn: this.a == v**; **ifOut: this.a == v**; **ifIn: this.a in s**; or **ifIn: this.a in s**; are added.
- 6) **Guards.** Guarded constraints **ifIn: c**; and **ifOut: c**; are replaced by equivalent constraints **this ->(c)**; and **!this ->(c)**; respectively.
- 7) **Aggregation with comprehension.** Eliminate the **children** or **selectedChildren** keywords as follows, assuming that  $c_1, \dots, c_k$  are the child features of the containing feature:

- Replace **avg(children.a)** by **sum(children.a) / count(children)**, and similarly for **selectedChildren**.
  - Replace **fct(children.a)** by **fct(c<sub>1</sub>.a, ..., c<sub>k</sub>.a)**, where **fct** is one of the aggregation functions **sum, mul, min, max, and, or, xor**.
  - Replace **count(children)** by the number of children of the containing feature.
  - Replace **count(selectedChildren)** by **sum((c<sub>1</sub> ? 1 : 0), ..., (c<sub>k</sub> ? 1 : 0))**.
  - Replace **fct(selectedChildren.a)** by **fct((c<sub>1</sub> ? c<sub>1</sub>.a : neut), ..., (c<sub>k</sub> ? c<sub>k</sub>.a : neut))**, where **fct** is one of the aggregation functions **sum, mul, and, or, xor**, and **neut** is the neutral element wrt. the aggregation function (i.e. 0 for addition, 1 for multiplication, true for conjunction, false for disjunction and xor). Remember that the **selectedChildren** keyword for these functions is only available if the parent decomposition enforces the selection of at least one feature.
- 8) **Relative names.** All relative names **parent, this** and **root** are resolved and replaced by unambiguous feature names.
  - 9) **Constraints.** A single constraint is obtained in TVL by removing all constraints and adding a single constraint to the root feature. This constraint is the conjunction of all previously removed constraints and corresponds to  $\Phi$ .
  - 10) **Decomposition operators.** First replace occurrences of **oneOf, allOf** and **someOf** by **group [1..1]**, **group [\*..\*]** and **group [1..\*]** respectively. In a second step, replace each occurrence of **\*** inside a cardinality by the number of child features.
  - 11) **Distributed definitions.** Gather feature definitions spread over different blocks into the single block inside the group statement of its parent.

This translation will effectively eliminate all constructs that are not in TVL<sub>NF</sub>.

## B. Semantics of TVL<sub>NF</sub>

The semantic domain defines the universe in which an element of the syntactic domain is to be interpreted. As in the existing definition by Schobbens *et al.* [8], the semantic domain is that of *product lines*, meaning that a given FM should be interpreted as a product line. In earlier definitions, a product line is formally defined as a *set of products*, and a product as a *set of features*. While this definition is still relevant in our case, it does not capture the notion of *attribute*. We thus redefine a product as a set of features that comes with a function providing a value for each attribute.

**Definition 5** (Semantic domain  $S$ ). The semantic domain of TVL, denoted  $S$ , is the set of all products, each product  $p$  being a couple  $p = (c, v)$  where  $c$  is a set of features and  $v$  is a valuation of the attributes, respecting  $\tau$ , formally:

$$S = \mathcal{P}(\mathcal{P}(N) \times \mathcal{P}(A \rightarrow \{int, real, bool, enum\}))$$

One could argue that the attributes should in fact not be part of the semantic domain, that they are just helpers to

express additional constraints between features. While this is a valid interpretation for FMs, it will preclude a number of possibilities offered by attributes, essentially further reasoning or filtering based on the values of the attributes (such as attribute optimisation [4], [11]).

Given a semantic domain, the semantic function describes how to interpret a specific element of the syntactic domain.

**Definition 6** (Semantic function  $\mathcal{M}$ ). *Given a TVL  $d \in \mathcal{L}_{TVL}$ , its semantics is given by the function*

$$\mathcal{M} : \mathcal{L}_{TVL} \rightarrow \mathcal{S},$$

where  $\mathcal{M}(d)$  is the set of all couples  $(c, v)$  with  $c \in \mathcal{P}(N)$  being a valid feature set and  $v : A \rightarrow \{int, real, bool, enum\}$  being a valid attribute valuation. Each  $(c, v) \in \mathcal{M}(d)$  is such that:

- $c$  contains the root:  $r \in c$ ;
- $c$  satisfies decomposition cardinality:

$$\forall f \in c \bullet \lambda(f) = \langle m..n \rangle \\ \Rightarrow m - o \leq |\{g | g \in c \wedge \omega(g) = 0 \wedge f \rightarrow g\}| \leq n - o \\ \text{where } o = |\{g | g \in N \wedge \omega(g) = 1 \wedge f \rightarrow g\}|;$$

- $c$  includes each selected feature's parent:

$$\forall g \in c \bullet f \rightarrow g \Rightarrow f \in c;$$

- $c$  and  $v$  satisfy  $\Phi$ , meaning that  $\llbracket \Phi \rrbracket c, v$  evaluates to true.

Note that in our definition, optionality has ‘priority over’ the decomposition relation, meaning that an optional child of an *and*-decomposed feature does *not* have to be included in a configuration if its parent is included. This corresponds to the intuitive meaning of optionality and gives rise to a number of semantics-preserving transformations as described by Czarnecki and Eisenecker [12]. For instance, if one child of an  $\langle 1..j \rangle$ -decomposed feature  $f$  is optional, then this is equivalent to all its children being optional, or to all its children being mandatory and  $f$  being  $\langle 0..j \rangle$ -decomposed.

**Definition 7** (Semantics of expressions  $\llbracket \cdot \rrbracket$ ). *The semantics of a Boolean expression over the set  $N$  of features and  $A$  of attributes is evaluated against a configuration  $c$  and a valuation  $v$ ; it is defined in Table III, where  $n \in N$  is a feature,  $a \in A$  is an attribute,  $d \in \mathbb{Z}$  is a integer,  $f \in \mathbb{R}$  is a real,  $t$  is an enum value.*

## VI. RELATED WORK

By far the most widely used notation in the literature is the graphical FM notation based on FODA [3]. Most of the subsequent proposals such as FeatuRSEB [13], FORM [6] or Generative Programming [12] only slightly modify this graphical syntax (e.g. by adding boxes around feature names).

One exception is Batory [14] who proposed the GUIDSL syntax, in which the FM is represented with a grammar. The GUIDSL syntax is further used as a file format of the feature-oriented programming tools AHEAD [14] and FeatureIDE [15]. The GUIDSL format is aimed at the engineer and is thus easy to write, read and understand. However, it does

TABLE III  
EXPRESSION SEMANTICS IN TVL AND TVL<sub>NF</sub>

$\llbracket \text{true} \rrbracket$	$= \text{true}$
$\llbracket \text{false} \rrbracket$	$= \text{false}$
$\llbracket n \rrbracket$	$= \text{true}$ iff $n \in c$
$\llbracket a \rrbracket$	$= v(a)$
$\llbracket a == t \rrbracket$	$= \text{true}$ iff $v(a)$ equals $t$
$\llbracket a != t \rrbracket$	$= \text{true}$ iff $v(a)$ does not equal $t$
$\llbracket E \text{ in } S \rrbracket$	$= \text{true}$ iff $\llbracket E \rrbracket \in \llbracket S \rrbracket$
$\llbracket n_1 \text{ requires } n_2 \rrbracket$	$= \text{true}$ iff $n_1 \notin c \vee n_2 \in c$
$\llbracket n_1 \text{ excludes } n_2 \rrbracket$	$= \text{true}$ iff $n_1 \notin c \vee n_2 \notin c$
$\llbracket B_1 \ \&\& \ B_2 \rrbracket$	$= \text{true}$ iff $\llbracket B_1 \rrbracket \wedge \llbracket B_2 \rrbracket$
$\llbracket B_1 \    \ B_2 \rrbracket$	$= \text{true}$ iff $\llbracket B_1 \rrbracket \vee \llbracket B_2 \rrbracket$
$\llbracket !B \rrbracket$	$= \text{true}$ iff $\llbracket B \rrbracket$ equals false
$\llbracket B_1 \ -> \ B_2 \rrbracket$	$= \text{true}$ iff $\llbracket B_1 \rrbracket \implies \llbracket B_2 \rrbracket$
$\llbracket B_1 \ <- \ B_2 \rrbracket$	$= \text{true}$ iff $\llbracket B_2 \rrbracket \implies \llbracket B_1 \rrbracket$
$\llbracket B_1 \ <-> \ B_2 \rrbracket$	$= \text{true}$ iff $\llbracket B_1 \rrbracket \Leftrightarrow \llbracket B_2 \rrbracket$
$\llbracket E_1 == E_2 \rrbracket$	$= \text{true}$ iff $\llbracket E_1 \rrbracket$ equals $\llbracket E_2 \rrbracket$
$\llbracket E_1 != E_2 \rrbracket$	$= \text{true}$ iff $\llbracket E_1 \rrbracket$ does not equal $\llbracket E_2 \rrbracket$
$\llbracket E_1 < E_2 \rrbracket$	$= \text{true}$ iff $\llbracket E_1 \rrbracket < \llbracket E_2 \rrbracket$
$\llbracket E_1 > E_2 \rrbracket$	$= \text{true}$ iff $\llbracket E_1 \rrbracket > \llbracket E_2 \rrbracket$
$\llbracket E_1 \leq E_2 \rrbracket$	$= \text{true}$ iff $\llbracket E_1 \rrbracket \leq \llbracket E_2 \rrbracket$
$\llbracket E_1 \geq E_2 \rrbracket$	$= \text{true}$ iff $\llbracket E_1 \rrbracket \geq \llbracket E_2 \rrbracket$
$\llbracket \text{and}(B_1, \dots, B_k) \rrbracket$	$= \text{true}$ iff $\bigwedge_{i \in [1, k]} \llbracket B_i \rrbracket$
$\llbracket \text{or}(B_1, \dots, B_k) \rrbracket$	$= \text{true}$ iff $\bigvee_{i \in [1, k]} \llbracket B_i \rrbracket$
$\llbracket \text{xor}(B_1, \dots, B_k) \rrbracket$	$= \text{true}$ iff $\bigoplus_{i \in [1, k]} \llbracket B_i \rrbracket$
$\llbracket E_1 + E_2 \rrbracket$	$=$ the value of $\llbracket E_1 \rrbracket + \llbracket E_2 \rrbracket$
$\llbracket E_1 - E_2 \rrbracket$	$=$ the value of $\llbracket E_1 \rrbracket - \llbracket E_2 \rrbracket$
$\llbracket E_1 / E_2 \rrbracket$	$=$ the value of $\llbracket E_1 \rrbracket / \llbracket E_2 \rrbracket$
$\llbracket E_1 * E_2 \rrbracket$	$=$ the value of $\llbracket E_1 \rrbracket * \llbracket E_2 \rrbracket$
$\llbracket -E \rrbracket$	$=$ the value of $-\llbracket E \rrbracket$
$\llbracket \text{abs}(E) \rrbracket$	$=$ the absolute value of $\llbracket E \rrbracket$
$\llbracket B_1 ? E_1 : E_2 \rrbracket$	$=$ if $\llbracket B_1 \rrbracket$ , then $\llbracket E_1 \rrbracket$ , otherwise $\llbracket E_2 \rrbracket$
$\llbracket \text{sum}(E_1, \dots, E_k) \rrbracket$	$=$ the value of $\sum_{i \in [1, k]} \llbracket E_i \rrbracket$
$\llbracket \text{mul}(E_1, \dots, E_k) \rrbracket$	$=$ the value of $\prod_{i \in [1, k]} \llbracket E_i \rrbracket$
$\llbracket \text{min}(E_1, \dots, E_k) \rrbracket$	$=$ the smallest value of the $\llbracket E_i \rrbracket$
$\llbracket \text{max}(E_1, \dots, E_k) \rrbracket$	$=$ the greatest value of the $\llbracket E_i \rrbracket$
$\llbracket \{ E_1, \dots, E_k \} \rrbracket$	$=$ the set $\{\llbracket E_i \rrbracket   i \in [1, k]\}$
$\llbracket [ d_1 .. d_2 ] \rrbracket$	$=$ the interval $[d_1, d_2]$
$\llbracket [ * .. d ] \rrbracket$	$=$ the interval $] -\infty, d[$
$\llbracket [ d .. * ] \rrbracket$	$=$ the interval $[d, +\infty[$
$\llbracket [ f_1 .. f_2 ] \rrbracket$	$=$ the interval $[f_1, f_2]$
$\llbracket [ * .. f ] \rrbracket$	$=$ the interval $] -\infty, f[$
$\llbracket [ f .. * ] \rrbracket$	$=$ the interval $[f, +\infty[$

not support arbitrary decomposition cardinalities, attributes, or the representation of the FM as a hierarchy.

Van Deursen and Klint [16] proposed the Feature Description Language (FDL), a textual language to describe features. FDL does not support attributes, cardinality-based decompositions, DAGs or duplicate feature names.

The SPLOT [17] and 4WhatReason [18] tools use the SXFM syntax and file format. While the format uses XML for metadata and the overall file structure, its representation of the FM is entirely text-based with the explicit goal to make it suitable for the engineer. It differs from the GUIDSL format in that it makes the tree structure of the FM explicit through (Python-style) indentation. It supports decomposition cardinalities but not attributes.

The feature modelling plugin [19] and the Fama framework [20] both use XML based file formats in which the whole FM is encoded in XML. These formats were not intended to be written or read by the engineer and are thus hard to interpret, mainly due to the overhead caused by XML tags and technical information that is extraneous to the model.

## VII. CONCLUSION

We argue that while graphical FM languages may be more intuitive and accessible to non-technical stakeholders, they are not always adapted to large FMs involving attributes and complex constraints. We propose TVL, a text-based variability modelling language using a C-like syntax. The goal of the language is to be *scalable*, by being *concise* and by offering mechanisms for *modularity*. TVL is also meant to be *comprehensive* so as to cover a wide range of FM dialects proposed in the literature. The TVL language targets IT professionals and contexts where graphical notations will not scale (or are inconvenient, as in an e-mail). We acknowledge that for non IT stakeholders or for informal discussions around the blackboard, graphical FMs might be more appropriate than TVL models. An advantage of text-based languages is that there are many well-accepted applications (viz. text editors, source control systems, diff tools, and so on) that support modelling out of the box. Furthermore, choosing a C-like syntax means lower learning curves for most software engineers. We hope that this will lead to an easier adoption of FMs in an industrial context.

At the moment, TVL is a language proposal, and we are requesting feedback from the variability modelling community. We developed a reference implementation for TVL in Java.<sup>2</sup> The library has two components; the *syntactic component* is a parser that performs type checking, checks well-formedness, and can normalize a model (eliminate syntactic sugar). Among other things, it can be used to implement TVL support in existing FM tools. The *semantic component* is able to translate a TVL file to either a Boolean CNF formula (if it does not contain numeric attributes), or to a CSP problem according to the formal TVL semantics.

## ACKNOWLEDGEMENTS

This work was partially funded by the Walloon Region, the Interuniversity Attraction Poles Programme, Belgian State, Belgian Science Policy under the MoVES project, the BNB and the FNRS.

## REFERENCES

- [1] Q. Boucher, A. Classen, P. Faber, and P. Heymans, "Introducing TVL, a text-based feature modelling language," in *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*, Linz, Austria, January 27-29. University of Duisburg-Essen, January 2010.
- [2] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [3] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," SEI, CMU, Tech. Rep., 1990.
- [4] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés, "Automated reasoning on feature models," in *Proceedings of CAiSE'05*, 2005.
- [5] D. Nestor, L. O'Malley, E. Sikora, and S. Thiel, "Visualisation of variability in software product line engineering," in *Proceedings of VaMoS'07*, 2007.
- [6] K. C. Kang, S. Kim, J. Lee, K. Kim, G. J. Kim, and E. Shin, "Form: A feature-oriented reuse method with domain-specific reference architectures," *Ann. Softw. Eng.*, vol. 5, pp. 143–168, 1998.

- [7] P. Tarr, H. Ossher, W. Harrison, and S. M. J. Sutton, "N degrees of separation: multi-dimensional separation of concerns," in *Proceedings of ICSE'99*, 1999.
- [8] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Feature Diagrams: A Survey and A Formal Semantics," in *Proc. of RE'06*, 2006.
- [9] P. Heymans, P.-Y. Schobbens, J.-C. Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen, "Evaluating formal properties of feature diagram languages," *IET Software Journal, Special Issue on Language Engineering*, vol. 2, no. 3, pp. 281–302, 2008.
- [10] D. Harel and B. Rumpe, "Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff," Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel, Tech. Rep., 2000.
- [11] T. T. Tun, Q. Boucher, A. Classen, A. Hubaux, and P. Heymans, "Relating requirements and feature configurations: A systematic approach," in *Proceedings of SPLC'09*, 2009.
- [12] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [13] M. L. Griss, J. Favaro, and M. d. Alessandro, "Integrating feature modeling with the rseb," in *Proceedings of ICSR'98*, 1998.
- [14] D. S. Batory, "Feature Models, Grammars, and Propositional Formulas," in *Proceedings of SPLC'05*, 2005.
- [15] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, "FeatureIDE: A tool framework for feature-oriented software development," in *Proceedings of ICSE'09*, 2009.
- [16] A. Deursen and P. Klint, "Domain-specific language design requires feature descriptions," *Journal of Computing and Information Technology*, vol. 10, p. 2002, 2002.
- [17] M. Mendonca, M. Branco, and D. Cowan, "S.P.L.O.T. - Software Product Lines Online Tools," in *Proceedings of OOPSLA'09*, 2009.
- [18] M. Mendonca, "Efficient reasoning techniques for large scale feature models," Ph.D. dissertation, University of Waterloo, 2009.
- [19] M. Antkiewicz and K. Czarnecki, "Featureplugin: Feature modeling plug-in for eclipse," in *Proceedings of the OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop*, 2004.
- [20] D. Benavides, S. Segura, P. Trinidad, and A. R. Cortés, "Fama: Tooling a framework for the automated analysis of feature models," in *Proceedings of VaMoS'07*, 2007.

<sup>2</sup>See the TVL website at <http://www.info.fundp.ac.be/~acs/tvl>.