

Introducing TVL, a Text-based Feature Modelling Language

Quentin Boucher, Andreas Classen,* Paul Faber and Patrick Heymans

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

Email: {qbo,acs,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.

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) [1]. 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.* [2]. 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 further decomposed. In addition, although not shown in the figure, each of the features has a *price*, which can be modelled as an attribute [3].

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.

Even though future tools might solve some of these issues, they would just attack the symptoms of the underlying problem. We attack its cause and propose to change the medium of the notation to just *plain text*. We do not question the need for a graphical representation, we rather propose an alternative and complementary textual notation. 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*.

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

TVL 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. 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 [4] and comes with a reference implementation available online.¹ 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, we survey related work in Section III and conclude in Section IV.

*FNRS Research Fellow

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

II. SYNTAX

In this section we present an overview of the TVL syntax using code snippets. The formal BNF grammar is available online. The different sub-sections introduce five major parts of the language i.e. features, attributes, expressions, constraints and modularisation mechanisms.

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

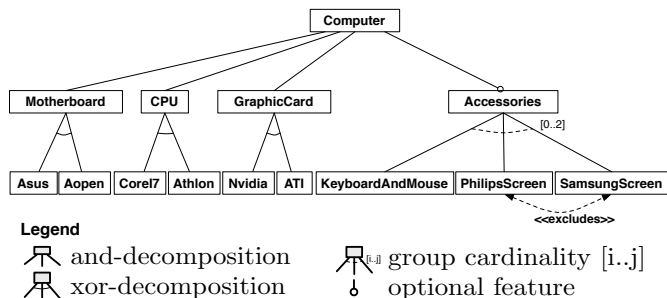


Fig. 1. Computer example FD

A. Feature hierarchy

The TVL language has a C-like syntax: it uses braces to delimit blocks, C-style comments and 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**. 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 [5]. DAG structures can also be modelled in TVL with the **shared** keyword associated to a feature name. 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**). Furthermore, in our example, 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. When declaring an attribute as an enumeration type, this means that it will take exactly one of a set of predefined values. The *socket*, for instance, is either *LGA1156* or *ASB1*.

```
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.

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 a fixed value for an attribute, e.g. *price* of *KeyboardAndMouse*. 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). 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 well as to express constraints on the FM. The language is strongly typed, each expression being either of type *bool*, *real* or *int*.

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.

D. Constraints

Constraints in TVL are attached to features. 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 example, the *socket* attribute of the *Motherboard* feature depends on the choice of the actual motherboard. 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;
    }
  }
}
```

E. Modularisation mechanisms

TVL offers various mechanisms that can help users to modularise large models. First of all, custom types can be defined at the top of the file and then be used in the FM. This allows to factor out recurring types. For instance, one might want to define the different sockets upfront and then use it as a type in an attribute declaration:

```
enum cpuSocket in {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;
```

One can also use 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);
```

Another mechanism is that features can be defined at one place and then extended further in the code. 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 [6]. For example, one could 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);
}

```

In this example, the decomposition of the root is defined at the beginning while its attributes are declared further down. The advantage of this is that the structure is easily understandable because it is not cluttered by attribute declarations.

III. RELATED WORK

By far the most widely used notation in the literature is the graphical FM notation based on FODA [2]. Most of the subsequent proposals such as FeaturSEB [7], FORM [5] or Generative Programming [8] only slightly modify this graphical syntax (e.g. by adding boxes around feature names).

One exception is Batory [9] 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 [9] and FeatureIDE [10]. The GUIDSL format is aimed at the engineer and is thus easy to write, read and understand. However, it does not support arbitrary decomposition cardinalities, attributes, or the representation of the FM as a hierarchy.

Van Deursen and Klint [11] 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 [12] and 4WhatReason [13] 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 [14] and the Fama framework [15] 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.

IV. CONCLUSION

We argue that while graphical FM languages may be more intuitive, they are not always adapted to large FMs involving attributes and complex constraints. We propose TVL, a text-based variability modelling language with 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. We acknowledge that for

non IT stakeholders or for informal discussions around the blackboard, graphical FMs might be more appropriate than TVL. 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 and evolution 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.² 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 defined in [4].

ACKNOWLEDGEMENTS

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

REFERENCES

- [1] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [2] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," SEI, CMU, Tech. Rep., 1990.
- [3] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés, "Automated reasoning on feature models," in *Proceedings of CAiSE'05*, 2005.
- [4] A. Classen, Q. Boucher, P. Faber, and P. Heymans, "Syntax and semantics of TVL, a text-based feature modelling language," PRECISE Research Centre, Univ. of Namur, Tech. Rep., 2010.
- [5] 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.
- [6] P. Tarr, H. Ossher, W. Harrison, and S. M. J. Sutton, "N degrees of separation: multi-dimensional separation of concerns," in *ICSE'99*, 1999.
- [7] M. L. Griss, J. Favaro, and M. d. Alessandro, "Integrating feature modeling with the rseb," in *Proceedings of ICSR'98*, 1998.
- [8] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [9] D. S. Batory, "Feature Models, Grammars, and Propositional Formulas," in *Proceedings of SPLC'05*, 2005.
- [10] 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.
- [11] A. Deursen and P. Klint, "Domain-specific language design requires feature descriptions," *Journal of Computing and Information Technology*, vol. 10, p. 2002, 2002.
- [12] M. Mendonca, M. Branco, and D. Cowan, "S.P.L.O.T. - Software Product Lines Online Tools," in *Proceedings of OOPSLA'09*, 2009.
- [13] M. Mendonca, "Efficient reasoning techniques for large scale feature models," Ph.D. dissertation, University of Waterloo, 2009.
- [14] M. Antkiewicz and K. Czarnecki, "Featureplugin: Feature modeling plug-in for eclipse," in *Proceedings of the OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop*, 2004.
- [15] 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.

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