A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL

Classen, Andreas; Boucher, Quentin; Heymans, Patrick

Published in:
Science of Computer Programming

Publication date:
2011

Document Version
Early version, also known as pre-print

Science of Computer Programming 76 (2011) 1130–1143

Contents lists available at ScienceDirect

Science of Computer Programming

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

A text-based approach to feature modelling: Syntax and semantics of TVL

Andreas Classen [*,1], Quentin Boucher, Patrick Heymans

PReCISE Research Centre, University of Namur, Rue Grandgagnage 21, B-5000 Namur, Belgium

ARTICLE INFO

Article history:
Received 5 March 2010
Received in revised form 5 August 2010
Accepted 21 October 2010
Available online 18 November 2010

Keywords:
Feature models
Code
Modelling
Language
Syntax
Semantics
Software product lines

ABSTRACT

In the scientific community, feature models are the *de-facto* standard for representing variability in software product line engineering. This is different from industrial settings where they appear to be used much less frequently. We and other authors found that in a number of cases, they lack concision, naturalness and expressiveness. This is confirmed by industrial experience.

When modelling variability, an efficient tool for making models intuitive and concise are feature attributes. Yet, the semantics of feature models with attributes is not well understood and most existing notations do not support them at all. Furthermore, the graphical nature of feature models' syntax also appears to be a barrier to industrial adoption, both psychological and rational. Existing tool support for graphical feature models is lacking or inadequate, and inferior in many regards to tool support for text-based formats.

To overcome these shortcomings, we designed TVL, a text-based feature modelling language. In terms of expressiveness, TVL subsumes most existing dialects. The main goal of designing TVL was to provide engineers with a human-readable language with a rich syntax to make modelling easy and models natural, but also with a formal semantics to avoid ambiguity and allow powerful automation.

1. Introduction

Software product line engineering (SPLE) is an increasingly popular software engineering paradigm which advocates systematic reuse across the software lifecycle. Central to the SPLE paradigm is the modelling and management of *variability*, i.e. *"the commonalities and differences in the applications in terms of requirements, architecture, components, and test artifacts"* [1]. Variability is typically expressed in terms of *features*, i.e. first-class abstractions that shape the reasoning of the engineers and other stakeholders [2]. Commercial (print on demand) printers, for instance, are developed as product lines and come with a broad range of features, such as support for spine captions, punching, or stapling. PRISMAprepare,[2] a commercial tool to prepare jobs for such printers, will serve as the running example in this paper.

A set of features can be seen as the specification of a particular product of the product line (PL). Feature models (FMs) [3,4] delimit the set of valid products of the PL. FMs are directed acyclic graphs, generally trees, whose nodes denote features and whose edges represent top-down hierarchical decomposition of features. The meaning of a decomposition link is that, if the parent feature is part of a product, then *some* of its child features have to be part of the product as well. Exactly which and how many of the child features have to be part depends on the type of the decomposition link. An excerpt of PRISMAprepare's FM is shown in Fig. 1, using the traditional graphical representation. The *and*-decomposition of the features *Document* and

* Corresponding author.
E-mail addresses: acs@info.fundp.ac.be (A. Classen), qbo@info.fundp.ac.be (Q. Boucher), phe@info.fundp.ac.be (P. Heymans).

1 FNRS research fellow.

2 See http://global.oce.com/products/prisma-prepare.

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.10.005

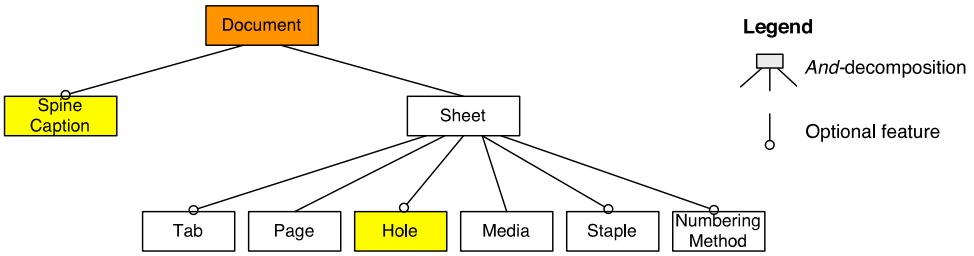

Fig. 1. Excerpt of PRISMAprepare's FM.

Sheet means that all their child features have to be selected, except for those that are optional, which is indicated by a hollow circle.

In the scientific community, FMs are the *de-facto* standard for representing the variability of an SPL. Several sources – our industry partners, discussions at the 2010 variability modelling (VaMoS) workshop [5] as well as recent literature reviews [6,7] – suggest that in the industrial world, in contrast, FMs appear to be used rarely.

One reason for this, we believe, is their lack of conciseness and naturalness when it comes to modelling realistic SPLs. Various industrial experiences have shown us that one of the most efficient tools for making FMs intuitive and concise are feature attributes [8], that is, typed parameters attached to features similar to attributes attached to classes in UML class diagrams. In the case of PRISMAprepare, for instance, the use of attributes reduced the number of features from 152 to 14, in another case one from 189 to 59 [9]. In the excerpt shown in Fig. 1, the colour of a feature indicates the number of attributes (yellow meaning *one* and orange meaning *two*) that the feature has. Discussions with engineers also showed that the resulting diagrams are more natural and easier to understand [9]. For instance, without attributes, we are forced to model alternative choices that are not further decomposed as a *xor*-decomposed feature group. A more concise solution would be to use an enumerated attribute. The semantics is exactly the same, but the notation is much more concise. In spite of all that, the semantics of FMs with attributes is not well understood and most existing notations and tools do not support them at all.

Another likely reason for the difficulty of using FMs in practice is the graphical nature of their syntax. Almost all existing FM languages are based on the FODA notation [3] which uses graphs with nodes and edges in a 2D space, as shown in Fig. 1. Feature attributes, to begin with, are intrinsically textual in nature and do not easily fit into this representation. Furthermore, *constraints* on the FM are often expressed as textual annotations using Boolean operators. If they were given a graphical syntax, attributes and constraints would only clutter a FM. When working with engineers, we also observed that a graphical syntax is a psychological barrier (having to draw models is deemed tedious and cumbersome by engineers) and poses a tooling problem. Existing tools for graphical FMs are generally research prototypes and are inferior in many regards to tool support for text-based formats (viz. text editors, source control systems, diff tools, no opaque file formats and so on).

To overcome these shortcomings, we designed TVL (Textual Variability Language), a text-based FM language. The idea of using text to represent variability in SPLE is not new [10,11] but seems to be recently gaining popularity [12,13]. In terms of expressiveness, TVL subsumes most existing dialects. The main goal of designing TVL was to provide engineers with a human-readable language with a rich syntax to make modelling easy and models natural, but also with a formal semantics to avoid ambiguity and allow powerful automation. Further goals for TVL were to be *lightweight* (in contrast to the verbosity of XML for instance) and to be *scalable* by offering mechanisms for structuring the FM in various ways.

Keeping with the tradition of the authors [4,14], TVL is defined formally. Its concrete, C-like, syntax is described by an LALR grammar, but it also has a mathematical abstract syntax and a denotational semantics. Having a well-defined tool-independent semantics further distinguishes TVL from most existing languages. A formal semantics is crucial for languages that are to be widely used or to serve as a format for information exchange. Moreover, it allows anyone to implement the language, serving as specification and reference. TVL is a *pure* language in the sense that all its constructs directly have a precise interpretation in the formal semantics. A reference implementation including a parser and a reasoning library is available online.[3]

The remainder of the paper is structured as follows. We survey related work in Section 2. Section 3 introduces TVL with code snippets from our running example. Section 4 gives well-formedness rules for TVL models while Section 5 specifies the formal semantics. We evaluate TVL in Section 6, followed by a description of our implementation in Section 7 and conclude in Section 8.

2. Related work

In the literature, graphical FM notations based on FODA [3] are by far the most widely used. Most of the subsequent proposals such as FeatuRSEB [15], FORM [16] or Generative Programming [17] are only slightly different from the original graphical syntax (e.g. by adding boxes around feature names).

But a number of textual FM languages were also proposed in the literature. Table 1 compares them. The criteria are (i) *human readability*, i.e. whether the language is meant to be read and written by a human; (ii) support for attributes;

[3] http://www.info.fundp.ac.be/~acs/tvl.

Table 1
Comparison of TVL to existing languages.

Language	Human readable	Attributes	Cardinalities	Basic Const.	Complex Const.	Structuring	Formal semantics	Tool support
FDL [10]	✓			✓			✓	
FMP [18]		✓	✓	✓		✓		✓
GUIDSL [19]	✓			✓				✓
FAMA [20]		✓	✓	✓	✓			✓
pure::variants [21]		✓	✓	✓	✓			✓
SXFM [22,23]	✓			✓				✓
CML [13]	✓	✓	✓	✓	✓			
VSL [24,12]	✓	✓	✓	✓				✓
KConfig[4]	✓	✓		✓		✓		✓
TVL	✓	✓	✓	✓	✓	✓	✓	✓

(iii) decomposition (group) cardinalities; (iv) basic constraints, i.e. *requires*, *excludes* and other Boolean constraints on the presence of features; (v) complex constraints, i.e. Boolean constraints involving values of attributes; (vi) mechanisms for structuring and organising the information contained in a FM (other than the FM hierarchy); (vii) formal and tool-independent semantics and (vii) tool support.

To our knowledge, the first textual language was FDL [10]. Apart from TVL, it is the only language for which a formal semantics exists. It does not support attributes, cardinality-based decomposition and other advanced constructs.

XML-based file formats to encode FMs are used by the Feature Modelling Plugin [18], the FAMA framework [20] and pure::variants [21]. 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. The semantics of FAMA and pure::variants is tool-based, given by the algorithms that translate an FM into SAT, CSP or Prolog. It is thus not readily accessible to an outsider.

Batory [19] proposed the GUIDSL syntax, in which the FM is represented by a grammar. The GUIDSL syntax is used as a file format of the feature-oriented programming tools AHEAD [19] and FeatureIDE [25]. 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.

The SPLOT [22] and 4WhatReason [23] 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 of being human-readable. 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.

Czarnecki [13] recently proposed the Concept Modelling Language (CML), a prototype language that is not yet fully defined. Its syntax resembles that of regular expressions, whereas TVL is closer to programming languages.

The CVM framework [24,12] supports text-based variability modelling with VSL which has support for many constructs. Attributes, however, can only be used as feature parameters and not in constraints.

KConfig is the configuration language of the Linux kernel. It is a configuration interface description language and a KConfig file can be interpreted as a FM. KConfig supports structuring with file includes. It only supports basic constraints which define presence of features.

We should note that all these languages are remotely related to constraint programming, and several implementations (including that of TVL) use constraint solvers internally. Moreover, as pointed out by Batory [19], FMs can be seen as simplified grammars where products correspond to sentences. Similarly, FMs with attributes such as TVL can be seen as a form of attribute grammar, albeit without the distinction of synthesised or inherited attributes [26,11]. What distinguishes FMs from constraint programming and attribute grammars is their domain-specific nature and independence from any of these technologies.

3. A guided tour of TVL

In this section we give an overview of the TVL syntax, illustrated using the PRISMAprepare FM introduced in Section 1. Information about the formal grammar can be found online[5] and in [27]. The following sub-sections introduce the major parts of the language: features, attributes, constraints and structuring mechanisms.

4 http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt.
5 http://www.info.fundp.ac.be/~acs/tvl.

The following TVL model is an excerpt of the PRISMAprepare FM.

```
1  // Declaring a custom type:
2  enum orientation in {horizontalLeft, horizontalRight, vertical};
3
4  // Declaring a structured type:
5  struct coord {
6     int x;
7     int y;
8  }
9
10 // The feature model:
11 root Document {
12    // And–decomposition of the root feature:
13    group allOf {
14       Sheet group [*..*] {
15          opt Tab, // an optional feature
16          Page,
17          opt Hole,
18          Media,
19          Staple,
20          NumberingMethod
21       },
22       opt SpineCaption
23    }
24
25    // Attribute declarations of the root feature:
26    enum type in {normal, booklet, perfectBinding};
27    enum stackMethod in {none, offset, mixed};
28
29    // A constraint:
30    Document.type == booklet -> !Sheet.Hole;
31 }
32
33 // The features SpineCaption and Hole are extended:
34 SpineCaption {
35    orientation orient;
36    ifIn: Document.type in {booklet, perfectBinding};
37 }
38
39 Hole {
40    coord position;
41 }
```

3.1. Feature declaration and hierarchy

We will ignore the first two declarations for the time being. The FM itself starts at line 11, with the declaration of the root feature. The feature hierarchy, graphically depicted in Fig. 1, follows on lines 11 through 23.

The root feature, *Document*, is decomposed into two sub-features by an *and*-decomposition: *Spine Caption* and *Sheet*. In TVL, each decomposition is introduced by the `group` keyword (line 13), which is followed by the decomposition type. The *and*, *or*, and *xor* decomposition types were renamed to `allOf`, `someOf` and `oneOf` in TVL. These names are inspired by [10] and make the language more accessible to people not familiar with the Boolean interpretation of decomposition. The decomposition type can also be given by a cardinality, as is done for the *Sheet* feature on line 14. Cardinalities can use constants, natural numbers, or the asterisk character (which denotes the number of children in the group). In our example, `group [*..*]` on line 14 is thus equivalent to `group [6..6]` or `group allOf`.

The decomposition type is followed by a comma-separated list of features, enclosed in braces. If a feature is optional, its name is preceded by the `opt` keyword (see, e.g., line 15). Each feature of the list can declare its own children, such as *Sheet* on line 14. If each feature lists its children this way, the tree structure of the FM will be reproduced in TVL with nested braces and indentation, as shown in Fig. 2(a). This can become a scalability problem for deep models, something we experienced in industrial cases. To this end, TVL allows one to declare a feature in the decomposition of its parent by just providing a name. A declared feature can then be extended later on in the code, as in Fig. 2(b).

Besides the `group` block, a feature can contain constraint and attribute declarations, all enclosed by a pair of braces. If there is only a `group` block, braces can be omitted. This reduces the number of braces in a pure decomposition hierarchy. To model a DAG structure (as in FORM [16]), a feature name can be preceded by the `shared` keyword, meaning that it is just a reference to a feature already declared elsewhere. (This is not illustrated in the example.)

```
root R group allOf {
   Level1 group allOf {
      Level2 group allOf {
         Level3 group allOf {
            ...
         }
      }
   }
}
```

(a) Reproduced in text.

```
root R group allOf {
   Level1 group allOf {
      Level2
   }
}
Level2 group allOf {
   Level3 group allOf {
      ...
   }
}
```

(b) Split up.

Fig. 2. Deep hierarchies can be split up in TVL.

```
Feature {
   int x;
}
```

(a) Basic.

```
Feature {
   int x in [0..10];
}
```

(b) Domain restriction.

```
Feature {
   int x is 10;
}
```

(c) Fixed value.

```
Feature {
   int x, ifIn: is 10,
          ifOut: is 0;
}
```

(d) Conditional value.

Fig. 3. Different ways of declaring an attribute in TVL.

3.2. Attributes

In our example, the *Document* feature has two attributes, indicating the types of document (line 26) and the stacking method (line 27) supported by the printer. Both attributes are of type `enum`, meaning that their value is one in a fixed set of values. This set of values is specified with the `in` keyword. The other attribute types supported by TVL are integer (`int`), real (`real`) and Boolean (`bool`). While Boolean and enumerated attributes could be encoded with features, this often results in unnecessary clutter of the diagram.

Attributes are declared like variables in C, in order to be intuitive for engineers. TVL further provides syntactic sugar to define the domain and the value of an attribute as illustrated in Fig. 3. If the value of an attribute depends on whether its parent feature is selected or not, the declaration shown in Fig. 3(d) can be used. Note that declarations (b), (c) and (d) could be equally expressed by declaration (a) followed by a constraint. However, syntactic sugar such as this allows the engineer to make models concise and to express her intentions clearly and intuitively.

Furthermore, to concisely specify cases in which the value of an attribute is an aggregate of another attribute that is declared for each child, an aggregation function can be used in combination with the `children` and `selectedChildren` keywords (followed by an `ID` denoting the attribute). This is not used in the case of PRISMAprepare, but a classical example is an attribute *price* declared for each feature. The attribute has a fixed value in the leaf features: `real price is 12.34;` and is calculated for all other features: `real price is sum(selectedChildren.price);`. Intuitively, this corresponds to a synthesised attribute in an attribute grammar.

3.3. Constraints

A constraint in PRISMAprepare is that sheets of a booklet may not be punched. This constraint is expressed at line 30 of the TVL model. As expected, the `==` denotes equality and `->` implication.

Constraints in TVL are Boolean expressions inside the body of a feature. There is also syntactic sugar for guarded constraints. For instance, if a document has a spine caption, it has to be a booklet or a perfect binding (for otherwise it is impossible to write on the spine of the document). This means that the domain of the enumerated *type* attribute is restricted if there is a spine caption. In TVL, this can be expressed with the guarded constraint at line 36. The `ifIn:` guard works just as for attributes: the constraint only applies if the parent feature, *SpineCaption*, is selected. While `ifIn:..` is equivalent to `SpineCaption -> ..`, it is more concise and requires less 'decoding' from the reader.

To facilitate specifying constraints and attribute values, TVL comes with a rich expression syntax. An overview is given in Table 2. The syntax is meant to be as complete as possible in terms of operators, to encourage writing of intuitive constraints. For instance, to restrict the allowed values of an enum, the set-style `in` operator can be used. For `enum e in {a, b, c, d, ..}`, the constraint `e in {b, c}` is syntactic sugar for `e != a && e != d && ..`, which is much less readable.

3.4. Structuring

TVL offers various mechanisms that can help engineers structure large models. For instance, custom types can be defined at the top of the file and then be used throughout the FM. This allows one to factor out recurring types and can thus reduce consistency errors. In the PRISMAprepare FM, the type *orientation* is defined on top (at line 2) as it appears in several places in the model (e.g. at line 35). Defining a custom type in this case increases the maintainability of the model as changes will be required only in the type declaration. It is also possible to define structured types to group attributes that are logically

linked. In the example, the type *coord* (at line 11) represents a pair of coordinates. It is used as a type for the *position* attribute of the *Hole* feature (at line 40) which represents the position of the hole.

There are two mechanisms for structuring. The first is the `include` statement, which takes as parameter a file path.

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

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. Modellers can thus structure the FM according to their preferences.

The second structuring mechanism, hinted at before, is that features can be defined in one place and be extended later in the code. Basically, a feature block may be repeated which adds constraints and attributes to the feature. In our example, the features *Hole* and *SpineCaption* are declared on lines 17 and 22, respectively. Their attributes and constraints are defined at lines 39–41 and 34–37, respectively. These could have been defined at lines 17 and 22, too, but this would have just cluttered the diagram.

These mechanisms allow modellers to organise the FM according to their preferences and can be used to implement separation of concerns [28]. This way, the engineer can specify the structure of the FM upfront, without detailing the features. Feature attributes and constraints can be specified in the second part of the file (as in our example), or in other files using the `include` statement.

4. Names, types and well-formedness rules

4.1. Naming, scope and references

The rules for naming 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.

Details about naming conventions can be found in [27]. The important point is that the scope of all feature names is global, but that feature names do not have to be globally unique. Only child features and attributes of the same feature must have distinct names. This allows for more flexibility when naming features and is very important when it comes to large industrial cases, where it is likely for features to have the same name. All feature references (inside constraints, for example) must still be unambiguous. To reference a feature with an ambiguous name, one can use a *qualified name*, that is the feature name prefixed by the names of its parents (separated by dots). A qualified name is unambiguous if the uppermost name is unambiguous. A model that contains ambiguous names is invalid and should be refused by a `TVL` implementation. Usage of qualified names can be helpful even for features with unique names. The parents in a qualified feature name give additional information about what the feature denotes, making it easier for the reader to understand the model.

The rules for referencing attributes are similar. Inside the body of the feature declaring them, they can be referenced solely with their name. Otherwise they have to be prefixed by the name of the feature that declared them. There are a number of keywords that 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. The `this` keyword is useful when referring to a child of the current feature with an otherwise ambiguous name. Use of these keywords has the advantage that the reader can immediately situate the referenced feature, without having to look up its name. This will make a model easier to understand.

4.2. Type correctness

`TVL` is strongly and statically typed, and does not allow casting. 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. Expressions involving attributes of type *enum* may only use enum values defined for the attribute.

4.3. Well-formedness rules

There are other rules a model must adhere to which are not enforced by the grammar. For instance, the grammar allows one to declare cycles in the decomposition relation or to have several `group` blocks per feature, both of which are not permitted. The restriction to a single `group` block is to ensure that a feature cannot be decomposed several times. Decomposition cardinalities $\langle i, j \rangle$ have to be so that $i \leq j$ and j has to be less or equal than the number of child features. 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 be 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 same rules apply to the `selectedChildren` keyword with the addition that it cannot be used for the `min` and `max` aggregation functions.

All the rules of this section should be checked by a TVL implementation. The next section assumes that a TVL model adheres to all of them.

5. Semantics

In line with previous work of the authors [4,14], a language is not fully defined without a formal semantics. Fortunately, part of the work has already been done elsewhere, mainly by Schobbens et al. [4] with the formal definition of Free Feature Diagrams (FFD), a parameterised FM language.

However, we cannot reuse the FFD definition as is. FFD are based on an abstract syntax that is much more limited than the concrete syntax of TVL. In Section 5.1, we thus define a translation from TVL to an abstract syntax close to that of FFD. Furthermore, FFD 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 is $\langle 0..1\rangle$-decomposed. We contribute these missing pieces in Section 5.2.

For the definition of the semantics, we follow the guidelines of Harel and Rumpe [29], meaning that we formally define the abstract syntax $\mathcal{L}$ of our language, the semantic domain $\mathcal{S}$ and the semantic function $\mathcal{M} : \mathcal{L} \to \mathcal{S}$.

5.1. Abstract syntax $\mathcal{L}_{TVL}$

The concrete syntax introduced in the previous section offers a number of syntactic shortcuts (structuring mechanisms, types,...). In order to obtain an easily formalisable language, the abstract syntax for TVL will be that of a *normal form* with fewer constructs but equal expressiveness.

Definition 1 (*TVL Abstract Syntax, Extension of [4]*)**.** The syntactic domain $\mathcal{L}_{\text{TVL}}$ is the set of all tuples $(N, r, DE, \omega, \lambda, A, \rho, \tau, V, \iota, \Phi)$ where:

- **N** is the (non-empty) set of features,
- $\mathbf{r} \in N$ is the root,
- $\mathbf{DE} \subseteq N \times N$ is the decomposition (hierarchy) relation between features. For $(n, n') \in DE$, n is the parent and n' the child feature. For convenience, we will sometimes write $n \to n'$ instead of $(n, n') \in DE$,
- $\omega : N \to \{0, 1\}$ labels optional features with a 1,
- $\lambda : N \to \mathbb{N} \times \mathbb{N}$ indicates the decomposition operator of a feature, represented as a cardinality $\langle i..j\rangle$ where i is 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,
- $\rho : A \to N$ is a total function that gives the feature declaring the attribute,
- $\tau : A \to \{int, real, enum, bool\}$ assigns a type to each attribute,
- **V** is the set of possible values for enumerated attributes,
- $\iota : \{a \in A | \tau(a) = enum\} \to \mathcal{P}(V)$ defines the domain of each enum,
- $\Phi \subseteq \mathcal{L}_{\text{exp}}$ is a set of Boolean-valued expressions over the features N and the attributes A, expressing additional constraints on the model. $\mathcal{L}_{\text{exp}}$ is the set of all correctly typed Boolean-valued expressions B that are formed according to the grammar given in Table 2, 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, t is an enum value and $v \in \mathbf{V}$ is an enum value.

Furthermore, each $d \in \mathcal{L}_{\text{TVL}}$ must satisfy the following well-formedness rules:

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

We recall that the abstract syntax, $\mathcal{L}_{\text{TVL}}$, only covers a subset of the concrete TVL syntax defined in Section 3. A TVL model using only constructs from $\mathcal{L}_{\text{TVL}}$ is in *normal form*, and the subset of the TVL language reduced to models in normal form is called TVL_{NF}. In the following, we will show that any TVL model can be transformed into an equivalent TVL_{NF} model. 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 and syntactic shortcuts that are not part of TVL_{NF} by giving a syntactic translation from TVL to TVL_{NF}. The second step is to define the semantics of TVL_{NF}, i.e. that of $\mathcal{L}_{\text{TVL}}$.

The concrete syntax of TVL_{NF} is a subset of the concrete syntax of TVL. The only allowed constructs are those defining the features and their hierarchy (N, r and DE), optional features (ω), cardinality-based decomposition operators (λ) and attributes with basic types (A, ρ and τ). The excluded constructs are mainly the structuring mechanisms and the non-cardinality decomposition operators. Furthermore, constraints in TVL_{NF} (Φ) have to be expressions of $\mathcal{L}_{\text{exp}}$. To obtain an

Table 2
Expression syntax $\mathcal{L}_{exp}$ of $\mathcal{L}_{TVL}$.

```
B ::= true | false | n | a | v | E in S |
      n excludes n | n requires n |
      B && B | B || B | !B |
      B -> B | B <- B | B <-> B |
      E == E | E != E |
      E <= E | E < E | E >= E | E > E |
      and(B [, B]*) | or(B [, B]*) |
      xor(B [, B]*)

E ::= n | a | t | d | q |
      E + E | E - E | E / E | E * E | - E |
      abs(E) | B ? E : E |
      sum(E [, E]*) | mul(E [, E]*) |
      min(E [, E]*) | max(E [, E]*)
S ::= { E [, E]* } |
      [ (d | *) .. (d | *) ] |
      [ (f | *) .. (f | *) ]
```

expression in $\mathcal{L}_{exp}$ from a TVL_{NF} expression, we have to define operator precedence, associativity and parentheses, since Table 2 abstracts away from these. We chose to define operator precedence in TVL and TVL_{NF} to be the same as in C (see [27]).

Now, the remaining constructs map 1:1 to the elements of $\mathcal{L}_{TVL}$ and $\mathcal{L}_{exp}$ in Definition 1. The first part of the semantics is provided in Definition 2 which specifies how to translate constructs that only exist in TVL into TVL_{NF}, thereby defining their semantics.

Definition 2. A model in TVL_{NF} is obtained from a model in TVL by applying the following transformation steps in the specified order.

1. **Includes.** Eliminate all `include` preprocessing directives by replacing them with the content of the referenced files.
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 t_t1; b2 t_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 one 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 one 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 keywords `children` and `selectedChildren` as follows, assuming that c_1, .., 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_1`.a, ..., `c_k`.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 feature.
   - Replace `count(selectedChildren)` by `sum((`c_1` ? 1 : 0), ..., (`c_k` ? 1 : 0))`.
   - Replace `fct(selectedChildren.a)` by `fct((`c_1` ? `c_1`.a : neut), ..., (`c_k` ? `c_k`.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 and *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 set of constraints is obtained in TVL by moving all constraints to the root feature.
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_{NF}.

5.2. Semantics of TVL_{NF}

The semantic domain defines the universe in which an element of the syntactic domain is to be interpreted [29]. As in the existing definition by Schobbens et al. [4], 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 3 (*Semantic Domain* $\mathcal{S}$)**.** The *semantic domain* of TVL, denoted $\mathcal{S}$, 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 τ and ι, formally:

$$\mathcal{S} = \mathcal{P}(\mathcal{P}(N) \times \mathcal{P}(A \to \mathbb{Z} \cup \mathbb{Q} \cup \{true, false\} \cup V))$$

Basically, with this definition, each attribute is treated like a variable that is always defined, even if the feature that declares it is not part of the product. We chose this interpretation as its flexible and emphasises the constraint-language aspect of FMs. An alternative interpretation would have been to assume that attributes of non-selected features do not exist (as if they were not declared). This would lead to several problems: what to do with attributes defined in terms of attributes that do not exist because their parents are not in the product, or: what is the semantics of constraints over undeclared attributes. Moreover, it would also cause problems when considering the semantics of FM configuration [30], where in an intermediate state some features are selected, some deselected and some undecided.

Given the semantic domain from Definition 3, the semantic function describes how to interpret each element of the syntactic domain from Definition 1.

Definition 4 (*Semantic Function* $\mathcal{M}$)**.** Given a TVL model $d \in \mathcal{L}_{\mathrm{TVL}}$, its semantics is given by the function $\mathcal{M} : \mathcal{L}_{\mathrm{TVL}} \to \mathcal{S}$, where $\mathcal{M}(d)$ is the set of all couples (c, v) with $c \in \mathcal{P}(N)$ being a *valid feature set* and $v : A \to \mathbb{Z} \cup \mathbb{Q} \cup \{true, false\} \cup V$ 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:

$$\begin{aligned} &\forall f \in c \bullet \lambda(f) = \langle m..n \rangle \\ &\quad \Rightarrow\ m - |opt_N| \leq |mand_c| \\ &\quad \wedge\ |all_c| \leq n \\ &\text{where: } opt_N = \{g | g \in N\ \wedge \omega(g) = 1\ \wedge f \to g\} \\ &\qquad\quad mand_c = \{g | g \in c\ \wedge \omega(g) = 0\ \wedge f \to g\} \\ &\qquad\quad all_c = \{g | g \in c\ \wedge f \to g\} \end{aligned}$$

- c includes each selected feature's parent:

$$\forall g \in c \bullet f \to g \Rightarrow f \in c$$

- c and v satisfy all the $\phi \in \Phi$, meaning that $\forall \phi \in \Phi \bullet [\![\phi]\!](c, v) \not\models false$. The semantics of an expression, $[\![\phi]\!](c, v)$, is quite standard and included for reference in [27].

While one might think that optional features are rather easy to formalise, the existing formal semantics by Schobbens et al. [4] only covers them indirectly (with syntactic preprocessing). Moreover, existing semantic discussions such as those by Czarnecki and Eisenecker [17] are limited to the interplay between optional features and standard *and*-, *or*- and *xor*-decompositions. As noted in [17], 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. A similar observation holds for a $\langle 1..1 \rangle$-decomposed feature with at least one optional child. This appears to cause confusion to the point that existing tools generally support optional features only as children in an *and*-decomposition.

Intuitively, optionality has 'priority over' the decomposition relation: an *and*-decomposition mandates that all features be included if their parent is, yet optional features are not bound by this requirement. Our definition generalises this intuition to the case of arbitrary $\langle i..j \rangle$ cardinalities. As can be seen in the second point of Definition 4, optional features cause the lower bound of a decomposition cardinality to decrease by the number of optional features opt_N. This alone would be incorrect; in addition, the features counted to satisfy the lower bound are only the mandatory features of the configuration $mand_c$. The latter part is best illustrated with an example, consider:

```
1    root f group [3..3] {
2       a, opt b, c
3    }
```

In that case, valid products are $\{f, a, b, c\}$ and $\{f, a, c\}$. If only the lower bound were decreased, $\langle 2..3 \rangle$, then the products $\{f, a, b\}$ and $\{f, b, c\}$ would be considered valid as well. This is why in Definition 4 the number of selected *mandatory* children of f has to be greater than the new lower bound.

The concept of feature attribute is also not formally defined in the existing literature. As discussed above, feature attributes exist independently of the feature that declares them. Our definition is purely declarative, it just requires that the attribute values satisfy all constraints. Such a definition lends itself well to implementation in SAT or CSP solvers. Furthermore, we chose not to fix attributes to a default value in case their parent feature is not part of the product. Basically, the same constraints apply to attributes whether their parent feature is selected or not (since the model is just one big constraint). Otherwise, it would be impossible to give fixed values to attributes (such as, the price of a feature). Furthermore, TVL provides appropriate syntactic sugar:

```
1  root f group allOf {
2     opt a {
3        int i, ifIn: in [1..10], ifOut: is 0;
4        int j is 42;
5        int k;
6     }
7  }
```

Here, the value of the attribute `i` is between one and ten if `a` is in the product, and zero otherwise. The attribute `j` is fixed at 42 and `k` can take any value.

6. Evaluation

Two evaluations of TVL were conducted: an empirical evaluation assessing TVL in industrial settings and a comparative evaluation assessing the relative strengths of TVL wrt. other FM notations on a large (also industrial) case.

6.1. Empirical evaluation

The empirical evaluation of TVL was conducted by a different team of authors [9]. We will briefly recall their research method and results. The research question addressed by this evaluation is: *"What are the benefits of TVL for modelling PL variability, as perceived by model designers and what are the PL variability modelling requirements that are not fulfilled by TVL?"*. This question was broken down into the set of language quality criteria given in Table 3.

The evaluation was carried out as a series of semi-structured interviews with practitioners from industry. Each interview was preceded by an analysis of an SPL from the participant's company, of which a TVL model was created in advance. Before each interview, the interviewee received an introduction to TVL followed by a walkthrough of the TVL model. The interviewees were then asked to rate TVL wrt. the quality criteria from Table 3, followed by an open discussion. The four companies (and their SPLs) were *GeezTeam* with *PloneMeeting*, *Océ Software Laboratories* with *PRISMAprepare*, *NXP Semiconductors* with a video processing unit and *Virage Logic* with *OSGeneric*.

The notation (criteria C1–C3 in Table 3) was generally well-received. The interviewees liked its simplicity and conciseness, and the compactness of attributes and constraints. They also noted the advantages of a textual, programming-like notation over a graphical or an XML-based language. While the structuring mechanisms (criterion C4) of TVL were considered an important feature of the language, and an advantage over other languages, the interviewees also noted the absence of more rigid modularisation mechanisms. Some of the suggested mechanisms were: modules that explicitly export features or attributes, inheritance between diagrams, and parameterised modules.

The expressiveness (criterion C5) was evaluated based on the TVL model created prior to the interview. To completely model the PloneMeeting case, TVL lacked string attributes; and for the PRISMAprepare case, it lacked the ability to express cloning of a feature. Everything else could be expressed, in the case of PloneMeeting more concisely than before. During the discussion, interviewees indicated other nice-to-have language features, such as: default values, optional attributes, string and date types, generic validators (e.g. for e-mail addresses), and error or warning messages that should be displayed

Table 3
Language evaluation criteria. The criteria are based on the programming language qualities from [31,32], for more details see [9].

C1	Clarity of notation	The meaning of constructs should be unambiguous and easy to read for non-experts.
C2	Simplicity of notation	The number of different concepts should be minimum. The rules for their combinations should be as simple and regular as possible.
C3	Conciseness of notation	The constructs should not be unnecessarily verbose.
C4	Modularisation	The language should support the decomposition into several modules.
C5	Expressiveness	The concepts covered by the language should be sufficient to express the problems it addresses. Proper syntactic sugar should also be provided to avoid convoluted expressions.
C6	Ease and cost of model portability	The language should be platform independent.
C7	Ease and cost of model creation	The elaboration of a solution should not be overly human resource-expensive.
C8	Ease and cost of model translation	The language should be reasonably easy to translate into other languages.
C9	Learning experience	The learning curve of the language should be reasonable.

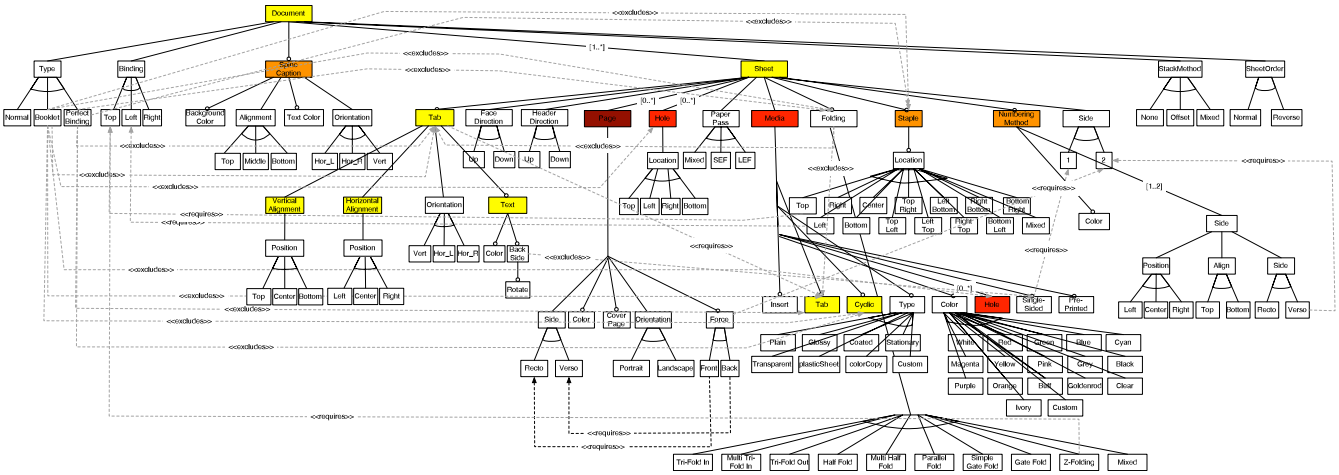

Fig. 4. Graphical FM of the PRISMAprepare printing options.

during configuration. It should be noted that these are features that also almost none of the existing FM languages supports. (Expressiveness wrt. other FM languages is evaluated in more detail in the next section.) The ease and cost criteria (C6–C8) were positively evaluated. Participants noted the impressive time gains for model creation compared to a graphical notation, as well as the implementation-independent semantics which facilitates model portability and translation. The learning curve (criterion C9) was universally judged to be gentle.

This sums up the principal results of the empirical evaluation. The interested reader is referred to [9] for further details.

6.2. Comparative evaluation

The empirical evaluation examines the qualities of TVL in isolation. A comparative evaluation was conducted to address the complementary research question: *What are the strengths and weaknesses of TVL wrt. existing textual FM languages?* From the languages of Table 1, the study compares FM languages (which excludes KConfig) that are textual and human readable (which excludes FMP, FAMA and pure::variants), and documented (which excludes CML). The FM representing the variability of the printing options of PRISMAprepare serves as the basis for the comparative study.

We created the initial FM of the PRISMAprepare printing options before the advent of TVL. The FM, shown in Fig. 4, is graphical and has 152 features. Fig. 4 is not meant to be readable,[6] it just illustrates the size of the FM. Grey lines are crosscutting constraints (requires and excludes) while the colour of a feature indicates the number of numeric attributes (from yellow meaning *one* to dark red meaning *six*) that the feature has. We subsequently modelled the printing options in FDL, GUIDSL, SXFM, VSL and TVL.[7]

Results. Table 4 shows the key statistics of these models (absence of a value denotes zero). Since TVL and VSL support enumerated and Boolean attributes, many features were replaced by such attributes. *Binding*, for instance has the children *top*, *left* and *right*. In a sense, these are not really features, but rather 'values' for the binding. In TVL or VSL, *binding* thus becomes an enumerated attribute of its parent feature, a change that is semantics-preserving. There are several such *xor*-decomposed features with leaf children only, and they contribute the most to the reduction in the number of features for the TVL and VSL models (138 features less).

Of the 69 attributes in the graphical FM, 37 are encoded with features and 32 are textual annotations (the coloured boxes). In TVL and VSL, no attributes are encoded with features. In FDL, GUIDSL and SXFM, the 32 numeric attributes could not be encoded and the other 37 attributes were encoded with features.

The way in which some concepts are modelled (attributes vs. features) has an impact on the way to express constraints on them. The graphical FM has a total of 42 constraints. Only half of them are actually graphical (requires/excludes) while the others are provided in the form of (TVL) annotations to the diagram. The TVL model encodes the same information with 23 constraints. Since FDL only has requires/excludes constraints, it cannot express 18 of the constraints. GUIDSL and SXFM allow arbitrary Boolean constraints over the features, which means that only the 11 constraints involving numeric attributes could not be represented. Observe that constraints in SXFM have to be in conjunctive normal form, which is not very intuitive to write or read. For this reason, there are no real requires/excludes constraints, and one of the more complex constraints was split in two (resulting in the number 32 rather than 31). Attributes in VSL are in fact parameterised features, and enumerated attributes are represented by features of type *string* with a fixed set of values. The semantics of such features, however, is undefined. While constraints can be written and appear to be syntactically correct in the CVM framework, they are not taken

6 Although readers of the electronic version can zoom in.

7 The models are available at http://www.info.fundp.ac.be/~acs/tvl/files/.

Table 4
Model statistics for the PRISMAprepare printing options FM.

	Graphical	TVL	FDL	GUIDSL	SXFM	VSL
Features	152	14	152	164	152	14
- encoded as attributes		37				37
- encoded as enum values		101				101
Attributes	69	69	37	37	37	69
- encoded as features	37		37	37	37	
- missing			32	32	32	
Constraints	42	34	24	31	32	34
- requires/excludes	24		24	24		
- Boolean constraints on features	7			7	32	
- on Boolean/enum attributes		23				
- on numeric attributes	11	11				
- missing constraints			18	11	11	34[8]
Depth	6	4	6	11	6	4

into account during integrity checking or configuration. Attributes in VSL are thus mere annotations and cannot be used for tasks that involve automated reasoning.

The depth of the graphical FM and of the FDL and SXFM models is six due to their encoding of attributes with features, while the depth of the TVL and VSL models is four. GUIDSL requires an additional parent feature in *and*-decompositions; it has thus 12 more features and a depth of 11.

The reason we choose to include a graphical FM in this evaluation is to illustrate (i) the blowup caused by not having enum and Boolean attributes and (ii) the graphical overhead of such large diagrams. There is, in fact, no graphical FM language with formal semantics that does support attributes. We thus chose to represent the attributes and constraints of the graphical FM with TVL. The ability of the FM to encode all of the constraints and numeric attributes is just because we chose to reuse TVL and its semantics for this purpose.

Observations. A first observation is that the lack of numeric attributes in the existing languages makes it impossible to model the whole case. The constraints here are only those inherent to the print options; the printer-related constraints include even more numeric constraints.

Another general observation is that the requirement of feature names to be unique (in FDL, GUIDSL and SXFM) results in cumbersome models, since the only sensible way to guarantee uniqueness in large models is to prefix each feature name with the names of its parents. In FDL this further leads to inconsistent prefixes, since it requires names of non-leaf features to start in uppercase. In our efforts to port the TVL model to FDL, GUIDSL and SXFM we noticed that even if some feature names are unique all by themselves, it is hard to identify them when referenced in a constraint by their name alone.

An advantage of TVL also appears to be its well-defined syntax and semantics. For VSL, for instance, there is – to our knowledge – no publicly available description of the syntax or the semantics. While the syntax can be mostly inferred from examples, the semantics can only be guessed. The same holds for SXFM, where the syntax has to be inferred by inspecting models created with the point-and-click interface.

6.3. Lessons learnt

The theory of cognitive fit [33] stipulates that the performance of an individual at a task depends on how well task and information representation match. In the domain of variability modelling in SPLE, there are a number of tasks ranging from informal discussions with stakeholders to actual formal model elaboration. Within this spectrum of activities, the cognitive fit of each FM representation will likely vary. TVL targets the end of this spectrum, when it comes to creating actual production FMs. The empirical evaluation largely confirms that a textual language has the best cognitive fit for these activities. Graphical FMs, on the other hand, probably offer a better cognitive fit at the beginning of the spectrum (discussions with stakeholders, or sketching). Furthermore, TVL does not preclude graphical visualisations which can be easily generated from a TVL model.

Let us revisit the arguments made in the introduction. The example clearly shows the necessity for attributes when modelling realistic cases. It also illustrates the advantages of enumerated attributes for achieving concise models. The comparative survey further confirmed that attributes are absent from other textual languages. The observation that the graphical nature of the syntax is a problem in industrial settings is confirmed by the empirical evaluation, especially in the

[8] Constraints involving attributes appear to be annotations with no semantics (see below).

reaction to criteria C3 and C7. Fig. 4 further illustrates the problem raised by the size of such a diagram. The evaluation also shows that design goals of the language were largely met: human-readability (beyond the fact of not being XML-based) is confirmed by positive feedback to criteria C1, C2 and C9; rich syntax and easy modelling were confirmed by positive feedback to criteria C5-8; and TVL being lightweight by criterion C3.

As for the expressiveness, the empirical evaluation showed that TVL can handle most of the industrial cases and the comparative evaluation confirmed that it is more expressive than existing textual languages.

Still, several shortcomings were pointed out. The main issue revealed during the empirical evaluation was the absence of more powerful modularisation mechanisms. A recurring remark was also the absence of attribute types such as *string* or *date*. While nothing technically prevents the addition of these types to the syntax, we explicitly left them out in our desire to keep the language *pure*. That is, we wanted TVL to have a concise and clear semantics that can be implemented in a straightforward way using declarative reasoners such as SAT or CSP. In consequence, we limited our data types to those generally used in SAT or CSP. We have since reconsidered our stance on this and will add support for those types to future revisions of TVL. To overcome the problem these types pose for declarative reasoners, they will only be allowed to appear in constraints that can be solved by a non-declarative preprocessor. Another recurring request were feature cardinalities to allow feature cloning (as in [17]). Again, the problem lies in defining a semantics for cloned features that is both intuitive and easy to implement. In absence of such a semantics, we preferred to leave feature cardinalities out of TVL for the time being.

Threats to validity of the empirical evaluation are discussed in detail in [9]. The authors judge that their *"results are valid for a wide range of organisations and products"* [9], since it is based on four different cases. Threats to the validity of the comparative study can stem from the way it was conduced and from the choice of the case. Concerning the former, most risks are mitigated by the process which consisted in translating a reference model (the graphical FM) into the various languages. This task does not require expertise of the modeller in the subject, only in the target language. Furthermore, it makes sure that comparisons between the produced models are meaningful, since all models refer to the same concepts. The risk posed by the expertise in the target languages was mitigated by consulting available documentation (although most languages are poorly documented) and by testing and validating the models with the respective tools. The choice of this particular case does not seem to threaten the generality of the conclusions: most observations we made are likely to be made on any model of similar size and complexity.

More anecdotal evidence of TVL's acceptance include the fact that it was well received at the 2010 variability modelling workshop [5,34] and that it was chosen as the FD language for the EU/FP7 project HATS[9] [35].

7. Implementation

Tool support for TVL exists in the form of a Java library available at the TVL website.[10] The library has two components. The *syntactic component* is a parser implemented with the CUP[11] parser generator. It performs all the checks discussed in Section 4 (references, types, well-formedness), as well as model normalisation as described in Section 5.1. Among other things, the syntactic component can be used to add TVL support to existing FM tools.

The *semantic component* of the library implements the semantics defined in Section 5.2. TVL models *without* numeric attributes are first normalised and then translated into a Boolean CNF formula as described in [19,36,23]. To analyse the model, the CNF formula is fed to the Sat4J[12] SAT solver. We emphasise that this translation can deal with the whole language except for numeric attributes, which means that it permits efficient FM analysis even in the presence of enumerated attributes. For TVL models *with* numeric attributes we use the CHOCO[13] CSP solver. The translation is very similar, except that now also constraints on numeric attributes are taken into account. The CSP part of the library is still in development. Although not the primary goal of the library, it is straightforward to implement analysers on top of it. A number of them already exist for checking satisfiability of an FM or validity of a product.

The library serves as a reference implementation for TVL in Java. The information given here and in [27] is sufficient to re-implement TVL from scratch. We are currently doing this using the term rewriting language ASF+SDF,[14] which is very natural for implementing model normalisation. This second implementation is unrelated to (and independent from) the Java library and is part of an ongoing experiment to compare the maintainability of the two parser technologies. The TVL grammar in the SDF format is also available online.

The Java library was used during the empirical evaluation in order to test syntactic correctness and integrity of the models. Being a library with a minimalistic front-end, it was not a subject of the empirical evaluation. Regarding the efficiency of FM analysis, most of the results of this extensive research area [37] do apply to TVL. In particular, the Java library uses SAT solving for analysing FMs (without numeric attributes), which was reported to be very efficient for FMs up to 10,000 features [38].

9 http://www.cse.chalmers.se/research/hats/.
10 http://www.info.fundp.ac.be/~acs/tvl.
11 http://www2.cs.tum.edu/projects/cup.
12 http://www.sat4j.org.
13 http://www.emn.fr/x-info/choco-solver.
14 http://www.meta-environment.org/.

8. Conclusion

We presented TVL, a textual FM language that targets IT professionals and contexts in which the cognitive fit of graphical notations is bad. TVL provides engineers with a human-readable language with a rich syntax that makes models more concise and natural. An important factor contributing to this, but neglected in most existing FM languages, are feature attributes. TVL supports attributes of various types and formalises the notion as part of its formal semantics. A further advantage of TVL, due to its text-based nature, 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. We hope that with these advantages, TVL contributes to a more widespread adoption of FMs in industrial contexts. A reference implementation of TVL in Java, with a full parser and support for FM analysis, is available online as open source.[15]

Acknowledgements

We thank our colleagues for their feedback on the language design, particularly Ebrahim Abbasi, Arnaud Hubaux, Raphaël Michel, Germain Saval and Pierre-Yves Schobbens. We also thank Paul Faber who implemented most of the TVL Java library and Anthony Cleve for his help with the ASF+SDF implementation. This work was partially funded by the Walloon Region under the ERDF and the NAPLES project, the IAP Programme, Belgian State, Belgian Science Policy under the MoVES project, the BNB and the FNRS.

References

[1] K. Pohl, G. Böckle, F.J. van der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer, 2005.
[2] A. Classen, P. Heymans, P.-Y. Schobbens, What's in a feature: A requirements engineering perspective, in: Proceedings of FASE'08, pp. 16–30.
[3] K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson, Feature-oriented domain analysis, FODA, feasibility study, Technical Report, SEI, CMU, 1990.
[4] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, Y. Bontemps, Feature diagrams: a survey and a formal semantics, in: Proc. of RE'06, pp. 139–148.
[5] D. Benavides, D. Batory, and P. Grünbacher (Eds.), Proceedings of VaMoS'10, ICB Research Report, vol. 37, Universität Duisburg-Essen, 2010.
[6] L. Chen, M.A. Babar, N. Ali, Variability management in software product lines: a systematic review, in: Proceedings of SPLC'09, pp. 81–90.
[7] A. Hubaux, A. Classen, M. Mendonca, P. Heymans, A preliminary review on the application of feature diagrams in practice, in: [5], pp. 53–59.
[8] D. Benavides, P.T. Martín-Arroyo, A.R. Cortés, Automated reasoning on feature models, in: Proceedings of CAiSE'05, pp. 491–503.
[9] A. Hubaux, Q. Boucher, H. Hartman, R. Michel, P. Heymans, Evaluating a text-based feature modelling language: four industrial case studies, in: 3rd International Conference on Software Language Engineeringm, SLE 2010 (in press).
[10] A. van Deursen, P. Klint, Domain-specific language design requires feature descriptions, J. Comput. Inf. Technol. CIT 10 (2002) 1–18.
[11] D. Batory, B.J. Geraci, Validating component compositions in software system generators, in: Proceedings of ICSR'96, pp. 72–81.
[12] A. Abele, Y. Papadopoulos, D. Servat, M. Törngren, M. Weber, The CVM framework — a prototype tool for compositional variability management, in: [5], pp. 101–106.
[13] K. Czarnecki, Variability modeling: state of the art and future directions (keynote), in: [5], p.11.
[14] P. Heymans, P.-Y. Schobbens, J.-C. Trigaux, Y. Bontemps, R. Matulevicius, A. Classen, Evaluating formal properties of feature diagram languages, in: Language Engineering, IET Softw. 2 (2008) 281–302 (special issue).
[15] M.L. Griss, J. Favaro, M.d. Alessandro, Integrating feature modeling with the RSEB, in: Proceedings of ICSR'98, pp. 76–85.
[16] K.C. Kang, S. Kim, J. Lee, K. Kim, G.J. Kim, E. Shin, Form: A feature-oriented reuse method with domain-specific reference architectures, Ann. Softw. Eng. 5 (1998) 143–168.
[17] K. Czarnecki, U.W. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
[18] M. Antkiewicz, K. Czarnecki, Featureplugin: feature modeling plug-in for eclipse, in: Proceedings of the OOPSLA'04 ETX Workshop.
[19] D.S. Batory, Feature models, grammars, and propositional formulas, in: Proceedings of SPLC'05, pp. 7–20.
[20] D. Benavides, S. Segura, P. Trinidad, A.R. Cortés, Fama: tooling a framework for the automated analysis of feature models, in: Proceedings of VaMoS'07, pp. 129–134.
[21] D. Beuche, Modeling and building software product lines with pure: :variants, in: SPLC'08, p. 358.
[22] M. Mendonca, M. Branco, D. Cowan, S.p.l.o.t. - software product lines online tools, in: Proceedings of OOPSLA'09, pp. 761–762.
[23] M. Mendonca, Efficient reasoning techniques for large scale feature models, Ph.D. Thesis, University of Waterloo, 2009.
[24] M.-O. Reiser, Core concepts of the compositional variability management framework (CVM), Technical Report, Technische Universität Berlin, 2009.
[25] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, S. Apel, Feature IDE: a tool framework for feature-oriented software development, in: Proceedings of ICSE'09, pp. 311–320.
[26] D.E. Knuth, Semantics of context-free languages, Math. Syst. Theory 5 (1971) 95–96.
[27] A. Classen, Q. Boucher, P. Faber, P. Heymans, The TVL specification, Technical Report, University of Namur, Belgium, 2010.
[28] P. Tarr, H. Ossher, W. Harrison, S.M.J. Sutton, N degrees of separation: multi-dimensional separation of concerns, in: Proc. ICSE'99, pp. 107–119.
[29] D. Harel, B. Rumpe, Modeling languages: syntax, semantics and all that stuff - part I: the basic stuff, Technical Report, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel, 2000.
[30] A. Classen, A. Hubaux, P. Heymans, A formal semantics for multi-level staged configuration, in: Proceedings of VaMoS'09, pp. 51–60.
[31] N. Holtz, W. Rasdorf, An evaluation of programming languages and language features for engineering software development, Eng. Comput. 3 (1988) 183–199.
[32] T. Pratt, Programming Languages : Design and Implementation, Prentice Hall, 1984.
[33] I. Vessey, Cognitive fit: A theory-based analysis of the graphs versus tables literature, Decis. Sci. 22 (1991) 219–240.
[34] Q. Boucher, A. Classen, P. Faber, P. Heymans, Introducing TVL, a text-based feature modelling language, in: [5], pp. 159–162.
[35] D. Clarke, N. Diakov, R. Hähnle, E.B. Johnsen, G. Puebla, B. Weitzel, P. Wong, Hats—a formal software product line engineering methodology, in: Proceedings of FMSPLE'10, co-located with SPLC 2010.
[36] C. Sinz, Towards an optimal cnf encoding of boolean cardinality constraints, in: Proceedings of CP'05, pp. 827–831.
[37] D. Benavides, S. Segura, A. Ruiz-Cortés, Automated analysis of feature models 20 years later: a literature review, Inf. Syst. 35 (2010) 615–636.
[38] M. Mendonca, A. Wasowski, K. Czarnecki, Sat-based analysis of feature models is easy, in: Proceedings of SPLC'09, pp. 231–240.

15 http://www.info.fundp.ac.be/~acs/tvl.