Evaluating a Textual Feature Modelling Language: Four Industrial Case Studies

Hubaux, Arnaud; Boucher, Quentin; Hartmann, Hermann; Michel, Raphaël; Heymans, Patrick

Published in:
Proceedings of the 3rd International Conference on Software Language Engineering (SLE'10). Eindhoven, The Netherlands, Collection LNCS, pp. 337-356

Publication date:
2010

Document Version
Early version, also known as pre-print

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

TECHNICAL REPORT August 3, 2010

AUTHORS | Arnaud Hubaux, Quentin Boucher, Herman Hartmann, Raphaël Michel, Patrick Heymans
APPROVED BY | P. Heymans
EMAILS | {ahu|qbo|phe}@info.fundp.ac.be, herman.hartmann@viragelogic.com, raphael.michel@cetic.be

STATUS | Submitted to SLE 2010
REFERENCE | P-CS-TR TVLEV-000001
PROJECT | MoVES
FUNDING | Interuniversity Attraction Poles Programme of the Belgian State of Belgian Science Policy, First DOC.A, European Regional Development Fund (ERDF)

Evaluating a Textual Feature Modelling Language: Four Industrial Case Studies

THE PRESENT DOCUMENT HAS BEEN SUBMITTED TO SLE 2010 AND IS CURRENTLY UNDER REVIEW. THE EVALUATION PART HAS OTHERWISE NOT BEEN PUBLISHED OR SUBMITTED ELSEWHERE.

Evaluating a Textual Feature Modelling Language: Four Industrial Case Studies

Arnaud Hubaux[1], Quentin Boucher[1], Herman Hartmann[3], Raphaël Michel[2], Patrick Heymans[1]

[1] PReCISE Research Centre, Faculty of Computer Science, University of Namur, Belgium
{ahu, qbo, phe}@info.fundp.ac.be
[2] CETIC Research Centre, Belgium
raphael.michel@cetic.be
[3] Virage Logic, High Tech Campus, The Netherlands
herman.hartmann@viragelogic.com

Abstract. Feature models are commonly used in software product line engineering as a means to document variability. Since their introduction, feature models have been extended and formalised in various ways. The majority of these extensions are variants of the original tree-based graphical notation. But over time, textual dialects have also been proposed. The "textual variability language" (TVL) was proposed to combine the advantages of both graphical and textual notations. However, the benefits and limitations of these notations have not been empirically evaluated up to now. In this paper, we evaluate TVL with four cases from companies of different sizes and application domains. The study shows that practitioners can benefit from TVL. The participants appreciated the notation, the advantages of a textual language and considered the learning curve to be short. The study also revealed requirements for feature modelling that were not covered by TVL.

1 Introduction

Feature models (FMs) were introduced as part of the FODA (Feature Oriented Domain Analysis) method 20 years ago [1]. They are a graphical notation whose purpose is to document variability, most commonly in the context of software product line engineering (PLE) [2]. Since their introduction, FMs have been extended and formalised in various ways (e.g. [3, 4]) and tool support has been progressively developed [5]. The majority of these extensions are variants of FODA's original tree-based graphical notation. Figure 1 presents an example of graphical tree-shaped FM modelling the variability of an eVoting component. The *and*-decomposition of the root feature (*Voting*) implies that all its sub-features have to be selected in a valid product. Similarly, the *or*-decomposition of the *Encoder* feature means that at least one of its child features has to be selected and the *xor*-decomposition of the *Default VoteValue* feature means that one and only one child has to be selected. Cardinality-based decompositions can also be defined, like *VoteValues* in the example. In this case, the decomposition type implies that at least two, and at most five sub-features of *VoteValues* have to be selected. Finally, two `<requires>` constraints impose that the feature corresponding to the default vote value is part of the available vote values.

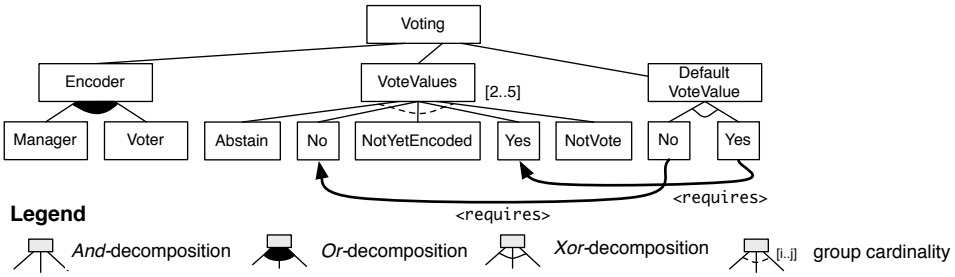

Fig. 1. FM of the PloneMeeting voting component.

Over time, textual dialects have also been proposed [6–9], arguing that it is often difficult to navigate, search and interpret large graphical FMs. The need for more expressiveness was a further motivation for textual FMs since adding constructs to a graphical language quickly starts harming its readability. Although advanced techniques have been suggested to improve the visualisation of graphical FMs (e.g. [10, 11]), these techniques remain tightly bound to particular modelling tools and are hard to integrate in heterogeneous tool chains [12]. Finally, our experience shows that editing functionalities offered by such tools are actually pretty limited and unhandy with large models.

Based on these observations, we proposed TVL [8, 13], a textual FM dialect geared towards software architects and engineers. Its main advantages are that (1) it does not require a dedicated editor—any text editor can fit—(2) its C-like syntax makes it both intuitive and familiar, and (3) it offers first-class support for modularity. However, TVL is meant to complement rather than replace graphical notations. It was conceived to help designers during variability modelling and does not compete, for instance, with graphical representations used during product configuration—which can actually be derived from it.

The problem is that empirical evidence showing the benefits and limitations of existing approaches, be they graphical or textual, is cruelly missing [8]. The goal of this paper is to collect evidence that demonstrates whether TVL is actually fit for practice, which is translated into the following research questions:

RQ1 *What are the benefits of TVL for modelling PL variability, as perceived by model designers?*

RQ2 *What are the PL variability modelling requirements that are not fulfilled by TVL?*

At this stage, it is important to understand that the goal of this research is neither to compare TVL to other languages, nor to assess capabilities of TVL other than its ability to model variability. Instead, this research aims at identifying the benefits and limitations of TVL, which can later be used for improving it or comparing it to other languages.

To answer these research questions, we conduct an *controlled field experiment* of TVL following a *sequential explanatory strategy* [14]. It consists in a *quantitative data analysis* followed by a *qualitative* one. The quantitative analysis is meant to collect data while the qualitative analysis assists in explaining the outcomes of the quantitative

analysis. Quantitative data is collected via evaluation forms based on a set of quality criteria inspired from the evaluation of programming languages. The main motivations for this are that frameworks for evaluating modelling languages are essentially dedicated to graphical rather than textual notations and that TVL is similar to a declarative constraint programming language. TVL was evaluated by five participants working for four companies of different sizes (from 1 to 28 000 employees), domains (hardware and software) and distribution strategies (proprietary and free software). Furthermore, TVL was evaluated by participants with different backgrounds in modelling and programming languages. Interviews during the qualitative analysis helped us (1) collect evidence that practitioners can benefit from TVL, (2) identify the types of stakeholders who can reap benefits from a language like TVL, and (3) elicit requirements not fulfilled by TVL.

The remainder of the paper is structured as follows. Section 2 looks into related work on feature-based variability modelling. Section 3 recalls the essence of TVL. Section 4 presents the research method and the four cases, whilst Section 5 presents the results of the quantitative and qualitative analyses. Section 6 analyses these results and Section 7 discusses the threats to validity.

2 Related Work

This section studies related work respectively dedicated to graphical and textual approaches to feature modelling.

2.1 Graphical feature models

Most common FM languages are graphical notations based on FODA which was introduced by Kang *et al.* [1] twenty years ago. Since this original proposal, several extensions have been proposed by various authors [15]. Mots of these graphical notations are meant to be accessible to non-technical stakeholders. However, working with large-scale FMs can become a tricky task with such notations. Given that a FM is a tree on a two dimensional surface, there will inevitably be large physical distances between features, which makes it hard to navigate, search and interpret them. Several tools have been developed to help modellers [16–19]. Most of them use directory tree-like representations of FMs to reduce physical distances between some features and provide collapse/expand functionalities. More advanced user interfaces and visualisation techniques have also been proposed to attenuate the aforementioned deficiencies (e.g. [10, 11]), but tools have their own drawbacks. First, building FMs can become a time consuming as they often require lots of clicks and drag-and-drops to create, place or edit elements. Second, they rely on software to visualise a model, meaning that without this software, like for instance on paper, blackboard or on a random computer, they will not work. Furthermore all those tools tend to have poor interoperability, which prevents effective collaborative work. Besides all those considerations, some constructs like inter-feature constraints or attributes cannot be easily accommodated into those graphical representations.

2.2 Textual feature models

Various textual FM languages have been proposed for a number of purposes. Their claimed advantages over graphical notations are: they do not require dedicated modelling tools and well-established tools are available for text-based editing, transformation, versioning... Furthermore, textual information and textual models can be easily exchanged, for instance by email.

To the best of our knowledge, FDL [6] was the first textual language. It is the only language to have a formal semantics. It also supports basic constraints and is arguably user friendly, but it does not include attributes, cardinality-based decompositions and other advanced constructs.

The AHEAD [7] and FeatureIDE [18] tools use the GUIDSL syntax [7], where FMs are represented through grammars. The syntax is aimed at the engineer and is thus easy to write, read and understand. However, it does not support decomposition cardinalities, attributes, hierarchical decomposition of FMs and has no formal semantics.

The SXFM file format is used by SPLOT [20] and 4WhatReason [21]. While XML is used for metadata, FMs are entirely text-based. Its advantage over GUIDSL is that it makes the tree structure of the FM explicit through indentation. However, except for the hierarchy, it has the same deficiencies as GUIDSL.

The VSL file format of the CVM framework [22, 23] supports many constructs. Attributes, however, cannot be used in constraints. The Feature Modelling Plugin [16] as well as the FAMA framework [24] use XML-based file formats to encode FMs. Tags make them hard to read and write by engineers. Furthermore, none of them proposes a formal semantics.

We can also mention the recently proposed Concept Modelling Language (CML) [9]. This language is still a prototype and is not yet fully defined or implemented.

3 TVL

Starting from the observation that graphical notations are not always convenient and that existing textual notations have limited expressiveness, formality and/or readability, we proposed TVL [8, 13], a textual alternative targeted to software architects and engineers. For conciseness, we can only recall here the basic principles of the language. More details about its syntax, semantics and reference implementation can be found in [13].

The following model will be used to illustrate some TVL constructs. It is a translation of the graphical model presented in Figure 1, which is an excerpt of the complete FM we built for PloneMeeting, one of the case studies. It captures the variability in the voting system that governs the discussion of meeting items[4]. Note here that the default vote value is here specified has an attribute rather than has a feature.

```
01 root Voting { // define the root feature
02  enum defaultVoteValue in {yes, no}; //attribute is either yes or no
03  (defaultVoteValue == yes) -> Yes; //yes requires Yes in VoteValues
```

[4] The complete model is available at `http://www.info.fundp.ac.be/~acs/tvl`

```
04  (defaultVoteValue == no) -> No; // no requires No in VoteValues
05  group allOf { // and-decomposition
06      Encoder { group oneOf {manager, voter} },
07      VoteValues group [2..*] { // <2..5> cardinality
08          Yes,
09          No,
10          Abstain
11          NotYetEncoded,
12          NotVote,
13      }
14  }
15 }
```

TVL can represent FMs that are either trees or directed acyclic graphs. The language supports standard decomposition operators [1, 3]: *or-* , *xor-*, and *and-*decompositions. For example, the *and*-decomposition of the `Voting` is represented from lines `05` to `12` (`group allOf {...}`). The *xor*-decomposition of the *Encoder* is represented at line `06` (`group oneOf {...}`). Generic cardinality-based decompositions [25] can also be defined using a similar syntax (see line `07`). Five different types of feature attributes [26] are supported: integer (`int`), real (`real`), Boolean (`bool`), structure (`struct`) and enumeration (`enum`). The domain of an attribute can be restricted to a predefined set of values using the `in` keyword. For instance, the set of available values of the enumerated attribute `defaultVoteValue` is restricted to `yes` and `no` (see line `02`). Attributes can also be assigned fixed or calculated values using the `is` keyword. Furthermore, the value of an attribute can differ when the containing feature is selected or not selected (`ifIn:` and `ifOut:` keywords). Several standard operators are available for calculated attributes (e.g. arithmetic operations). Their value can also be computed using aggregation functions over lists of attributes. Calculated attributes are not illustrated in the example.

In TVL, constraints are attached to features. They are Boolean expressions that can be added to the body definition of a feature. The same guards as for attributes are available for constraints. They allow to enable (resp. disable) a constraint depending on the selection (resp. non-selection) of its containing feature. Line `05` of the example is an example of (unguarded) constraint where the assignment of the `yes` value to `defaultVoteValue` attribute requires the selection of the `Yes` feature.

TVL offers several mechanisms to reduce the size of models and modularise them. We only touch upon some of them here and do not illustrate them in the example. First, custom types can be defined and then used in the FM. This allows to factor out recurring types. It is also possible to define structured types to group attributes that are logically linked. Secondly, TVL supports constants that can be used inside expressions or cardinalities. Thirdly, include statements can be used to add elements (e.g. features or attributes) defined in an external file anywhere in the code. Modellers can thus structure the FM according to their preferences. The sole restriction is that the hierarchy of a feature can only be defined at one place (i.e. there is only one group structure for each feature). Finally, features can have the same name provided they are not siblings.

Qualified feature names must be used to reference features whose name is not unique. Relative names like `root`, `this` and `parent` are also available to modellers.

4 Research Method

This section describes the settings of the evaluation, the goals and the four cases along with the profiles of the companies and participants involved in the study. The section ends with a description of the experiment's protocol.

4.1 Objectives

The overall objective of this paper is to evaluate the ability of TVL to model the variability of a product line (PL) as perceived by software engineers. The criteria that we use to measure the quality of TVL are inspired and adapted from [27, 28]. Originally, these criteria were established to evaluate the quality of programming languages. We have selected programming language criteria because prevailing modelling language evaluation frameworks like [29] are primarily designed for graphical notations. Additionally, TVL somehow resembles a declarative constraint programming language whose constructs are tailored to variability modelling. Finally, TVL should ultimately be integrated in development environments like eclipse or advanced text editors like emacs or vim. TVL is thus likely to be assimilated to a programming language by developers. We outline the quality criteria relevant to our study below.

Quality criteria adapted from [27, 28]	
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.

Note that given the explanatory nature of the study, no hypothesis was made, except that the participants had the appropriate expertise to answer our questions.

4.2 Cases

The evaluation of TVL was carried out with five participants coming from four distinct companies. Table 2 summarises the profiles of the five participants involved in the evaluation as well as the company and project they work for. For each participant, we indicate his position, years of experience in software engineering, his fields of expertise, the modelling and programming languages he used for the last 5 years, his experience with PLE and FMs, and the number of years he actively worked on the selected project. Note that for the experience with languages, PLE and FMs, we also mention the frequency of use, i.e. *intensive*/*regular*/*casual*/*evaluation*.

PloneMeeting

Description PloneGov [30] is an international Open Source (OS) initiative coordinating the development of eGovernment web applications. PloneGov gathers hundreds of public organizations worldwide. This context yields a significant diversity, which is the source of ubiquitous variability in the applications [31–33]. We focus here on PloneMeeting, PloneGov's meeting management project developed by GeezTeem. PloneMeeting is built on top of Plone, a portal and content management system (CMS) written in Python.

The current version of PloneMeeting extensively uses UML for code generation. However, the developers are not satisfied by the limited editing functionalities and flexibility offered by their UML tool. To eliminate graphical UML models, they developed *appy.gen*[5]. *appy.gen* allows to encode class and state diagrams, display parameters, portlet creation and consistency rules. In a nutshell, *appy.gen* enables the automated generation of full-blown Plone applications.

PloneMeeting is currently being re-engineered with *appy.gen*. A major challenge is to extend *appy.gen* to explicitly capture variation points and provide systematic variability management. We collaborate with the developers to specify the FM of PloneMeeting.

Participant We interacted with several developers during the collaboration, but only the main developer provided the quantitative and qualitative feedback for this evaluation. The initial motivation of the developer to engage in the evaluation was to assess the opportunity of using FMs for code generation. He has not used FMs to that end so far because, in his words, *"graphical editing functionalities (typically point-and-click) offered by feature modelling tools are cumbersome and counter-productive"*. The textual representation of FMs is therefore more in line with his development practice than their graphical counterpart.

[5] Available online at `http://appyframework.org/gen.html`

Table 2. Profiles of the five participants.

Criteria	**PloneMeeting**	**PRISMAprepare**	**CPU calculation**	**OSGeneric**	
Company	*GeezTeem*	*OSL Namur S.A.*	*NXP Semiconductors*	*Virage Logic*	
#Employees	1	70	28 000 (worldwide)	700 (worldwide)	
Location	Belgium	Belgium	The Netherlands	The Netherlands	
Type of software	Open source	Proprietary	Proprietary	Proprietary	
Project kickoff date	January 2007	May 2008	June 2009	2004	
Project stage reached	Production	Production	Development	Production	
Model version	1.7 build 564	4.2.1	July 6, 2009	September 30, 2009	
Position	Freelance	Product Line Manager	Senior Scientist	Senior Software Architect	Development Manag
Years of experience in SE	12+ years	31+ years	20+ years	15+ years	20+ years
Fields of expertise	WA, CMS	PM	PM, SPI, SR	McS, RPC, RT	ES, RT
Modelling languages	UML (regular)	None	UML (casual), DSCT (evaluation)	UML (casual)	UML (regular)
Programming languages	Python (intensive), Javascript (regular)	C++ (casual)	C (casual), Prolog (regular), Java (evaluation)	C (intensive)	C (casual), C++ (c al), Visual Basic (c al), Python (casual)
Experience with PLE	2 years (evaluation)	1 year (casual)	3 years (intensive)	4 years (regular)	4 years (intensive)
Experience with FDs	2 years (evaluation)	1 year (casual)	3 years (intensive)	4 years (casual)	4 years (intensive)
Project participation	3 years	2 years	1 year	6 years	2 years

Legend

CMS - Content Management System, **ES** - Embedded System, **McS** - Multi-core System, **PM** - Project Management, **RPC** - Remote Procedure Call, **RT** - Real Time, **SE** - Software Engineering, **SPI** - Software Process Improvement, **SR** - Software Reliability, **WA** - Web Applications

PRISMAprepare

Description Océ Software Laboratories S.A. (OSL) [34], is a company specialized in document management for professional printers. One of their main PLs is *Océ PRISMA*, a family of products covering the creation, submission and delivery of printing jobs. Our collaboration focuses on one sub-line called PRISMAprepare, an all-in-one tool that guides document preparation and whose main features are the configuration of printing options and the preview of documents.

In the current version of PRISMAprepare, mismatches between the preview and the actual output can occur, and, in rare cases, documents may not be printable on the selected printer. The reason is that some incompatibilities between document options and the selected printer can go undetected. For example, a prepared document can require to staple all sheets together while the target printer cannot staple more than 20 pages together, or does not support stapling at all. The root cause is that only some constraints imposed by the printers are implemented in the source code, mostly for time and complexity reasons.

Consequently, OSL decided to enhance its PL by automatically generating configuration constraints from printing description files. The objective is twofold: (1) build complete constraint sets and (2) avoid the manual maintenance of cumbersome constraints. So far, we have designed a first FM of the variability of the printing properties.

Participant Feedback was provided by the *product line manager* of PRISMAprepare. OSL is currently evaluating different modelling alternatives to express the variability of its new PL and generate the configuration GUI.

CPU calculation

Description NXP Semiconductors [35] is an international provider of Integrated Circuits (IC). ICs are used in a wide range of applications like automative, infotainment or navigation systems. They are the fundamental pieces of hardware that enable data processing like video or audio streams. ICs typically embed several components like CPU, memory, input and output ports, which all impose constraints that condition their combinations.

In this paper, we focus on the FM that models the variability of a video processing unit and study the impact it has on the CPU load. The FM, which is still under development, is meant to be fed to a software configurator. Thereby, it will support the customer during the selection of features while ensuring that no hardware constraint be violated (e.g. excessive clock speed required by the features). The FM also allows the user to strike an optimal price/performance balance, where the price/performance ratio is computed from attributes attached to features and monitored within the configuration tool.

Participant The evaluation was performed by the developer who originally created the FM with the pure::variants [17] tool, before and independently from this experiment. Prolog was used for defining the constraints. The major problem is that the time needed to implement the calculation over attributes was deemed excessive compared to the time needed to design the whole FM. This lead the company to consider TVL as an alternative.

OSGeneric

Description Virage Logic [36] is a supplier of configurable hardware and software to a broad variety of customers such as the Dolby Laboratories, Microsoft and AMD. Its variability-intensive products allows its customers to create a specific variant for the manufacturing of highly tailored systems on chip (SoC). OSGeneric (Operating System Generic) is a PL of operating systems used on SoCs. The produced operating systems can include both proprietary and free software. Each SoC can embed a large variety of hardware and contain several processors from different manufacturers.

Participants The evaluation was performed by two participants: the lead software architect of OSGeneric and the software development manager. Their PL is currently modelled with pure::variants. The participants are now considering other techniques in modelling the variability. Their motivation for evaluating TVL lies in using a language that (1) is more suited for engineers with a C/C++ background, (2) has a lower learning curve than pure::variants and (3) makes use of standard editors.

4.3 Experiment protocol

In this experiment, TVL was evaluated through interviews with the five participants of the four companies. Interviews were conducted independently from each other, except for Virage Logic where the two participants were interviewed together. Two researchers were in charge of the interviews, the synthesis of the results and their analysis. For each interview, they followed the protocol presented in Figure 2.

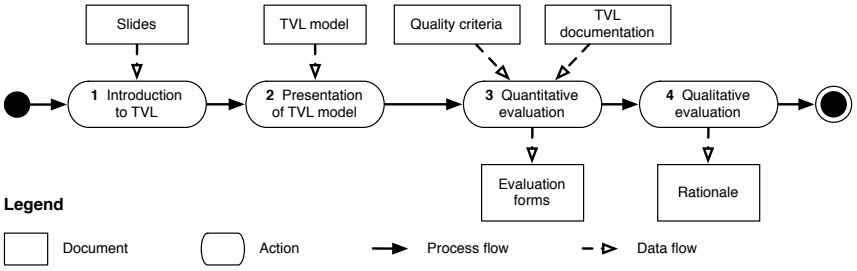

Fig. 2. Interview protocol.

The protocol starts with a short introduction to TVL (*circa* 20 minutes) that aims at giving the participants an overview of the language. At this stage, the participants are not exposed to details of the language, which resembles the setting of an out-of-the-box experience [37]. The goal of the second step is to provide the participants with a real TVL model. The appointed researchers designed, for each company and prior to the interviews, TVL models that respectively correspond to the configuration menus of PloneMeeting and PRIMSAprepare and the FMs of the CPU calculation and OS-Generic. The presentation of the TVL model was limited to 30 minutes to keep the

participants focused on the understanding of the model and avoid untimely discussions about the quality of the language. After the participants have been exposed to more details of the language and how their PL is modelled using TVL, they are able to form a judgement on its quality.

During the third step, the participants fill out the evaluation form presented in Table 3. The evaluation scale proposed to the participants is: ✚ the participant is strongly satisfied; **+** the participant is rather satisfied; ◯ the participant is neither satisfied nor unsatisfied; **-** the participant is rather unsatisfied; ▬ the participant is completely unsatisfied; N/A the participant is not able to evaluate the criterion.

The results of the evaluation are then discussed during the fourth step of the protocol. The qualitative information collected during this last phase is obtained by asking, for each criteria, the rationale that drove the participant to give his mark. On average, these two final steps lasted two hours.

5 Results

Table 3 synthesises the evaluation of TVL performed by the participants of GeezTeem, OSL, NXP and Virage Logic. Note that we kept the evaluations of the two Virage Logic participants separate, has indicated by the two columns under OSGeneric.

Table 3. Results of the evaluation of TVL.

Criterion		**PloneMeeting**	**PRISMAprepare**	**CPU calculation**	**OSGeneric**	
C1	Clarity of notation	+	+	◯	✚	+
C2	Simplicity of notation	+	✚	+	✚	+
C3	Conciseness of notation	✚	✚	+	✚	✚
C4	Modularisation	◯	+	+	+	+
C5	Expressiveness	-	◯	✚	✚	✚
C6	Ease and cost of model portability	+	✚	✚	✚	✚
C7	Ease and cost of model creation	+	+	✚	◯	◯
C8	Ease and cost of model translation	✚	✚	+	◯	+
C9	Learning experience	+	✚	+	+	+

To facilitate the explanation, we group the criterion into five categories: *notation*, *modularisation*, *expressiveness*, *ease and cost*, and *learning experience*. Note that, the collaborations with OSL, NXP and VirageLogic are protected by non-disclosure agreements. Therefore, specific details of the models are not disclosed.

Notation [C1-C3]. The participant unanimously appreciated the notation and the advantages of the textual notation in facilitating editing of the model (creating, modifying and copy/pasting model elements). The NXP and VirageLogic participants liked the compactness of attributes and constraints and the fact that attributes were explicitly part of the language rather than an add-on plugged onto a graphical notation.

The GeezTeem participant appreciated the ability of the language to express constraints very concisely. He reports that *appy.gen*, his website generator, offers two major ways of specifying constraints. First, guards can be used to make the value of an attribute depend on the value of another attribute. Secondly, Python methods can be used to express arbitrary constraints. These mechanisms can rapidly lead to convoluted constraints that are hard to maintain and understand. Additionally, developers struggle to maintain these constraints across web pages. The participant reports that at least 90% of the constraints (both within and across pages) implemented in classical *appy.gen* applications could be more efficiently expressed in TVL.

The OSL participant was particularly satisfied to see that TVL is not based on XML. He reported that their previous attempts to create XML-based languages were not very satisfactory because of the difficulty to write, read and maintain them. He also reported that the model represented in the language is much more compact than anything he could have produced with existing graphical representations.

The NXP participant was concerned about the scalability of the nested structure, i.e. the tree-shaped model, offered by TVL. He also reports that people used to graphical notations who already know FMs might prefer classical decomposition operators (*and*, *or*, *xor*) rather than their TVL counterparts (`allOf`, `someOf`, `oneOf`). Finally, the participants from NXP and Virage Logic were confused by the fact that the `->` symbol can always replace `requires` but not the other way around. In their opinion, a language should not offer more than one means to express the same thing.

One of the Virage Logic participants reports that attributes might be hard to discern in large models. He suggested to declare them in an Interface Description Language (IDL) style by prefixing the attribute declaration with the `attribute` keyword.

Modularisation [C4]. The ability to define a feature at one place and extend it further in the code was seen as an undeniable advantage as it would allow to decompose the FM among developers. The Virage Logic participants both discussed the difference between the TVL `include` mechanism and an *import* mechanism that would allow to specify exactly what part of an external TVL model can be imported but also what parts of a model can be exported. In their opinion, it would improve FM modularisation and module reuse since developers are already used to import mechanisms.

Apart from the `include` mechanism, TVL does not support model specialisation and abstraction (in an object-oriented fashion). In contrast, the developer of *appy.gen* considers them as fundamental. Their absence is one of the reasons that lead them to drop UML tools. Along the same lines, the OSL participant argued that the `include` should be augmented to allow *macro* definitions. By *macro*, the participant meant parametrized models similar to parametrized types, e.g. Java generics. A typical use case of that mechanism would be to handle common modelling variability patterns.

Expressiveness [C5]. The Geezteem participant expressed that TVL is sufficiently expressive to model variability in most cases. However, he identified several constructs missed by TVL that would be needed to model PloneMeeting. First, TVL does not offer *validators*. In his terms, a validator is a general-purpose constraint mechanism that can constrain the formatting of a field. For instance, validators are used to specify the elements that populate a select list, to check that an email address is properly formatted or that a string is not typed in where the system expects an integer. Secondly, he makes intensive use of the specialisation and abstraction mechanisms available in object-oriented, which are not available. These mechanisms are typically used to refine already existing variation points (e.g. add an attribute to a meeting item) or to specify abstract variation points that have to be instantiated and extended when generating the configuration menu (e.g. an abstract meeting attendee profile is built and it has to be instantiated before being available under the *Encoder* feature in Figure 1). Thirdly, multiplicities are used to specify the number of instances, i.e. clones, of a given element. Cloning is a fundamental aspect of *appy.gen* as many elements can be cloned and configured differently in Plone applications. This corresponds to feature cardinalities, which have already been introduced in feature modelling [3], but are currently not supported in TVL. Besides offering more types of attributes, *appy.gen* also allows developers to add parameters to attributes, e.g., to specify whether a field can be edited or requires specific read/write permissions. Type parameters are mandatory in *appy.gen* to support complete code generation. Finally, in order to be able to display web pages in different languages, *i18n* labels are attached to elements. *i18n* stands for internationalisation and is part of Plone's built-in translation management service. Translations are stored in key/value pairs. A key is a label in the code identifying a translatable string; the value is its translation. For instance, the `meeting_item_i18n` element will be mapped to *Meeting Item* (English version) and *Point de discussion* (French version). In most cases, several labels are attached to an element (e.g. its human-readable name and a description text).

The OSL participant also pointed out some missing constructs in TVL. First, default values which are useful in their projects for things like page orientation or paper dimensions. Secondly, feature cloning is missing. In PRISMAprepare, a document is normally composed of multiple *sheets*, where sheets can be configured differently and independently from one another. Thirdly, optionality of attributes should be available. For instance, in TVL, the *binding margin* of a page was specified as an attribute determining its size. If the document does not have to be bound, the *binding margin* attribute should not be available for selection.

The NXP and VirageLogic participants also recognized that feature cloning and default features were missing in the language. Additionally, they miss the specification of error, warning and information messages directly within the TVL model. These messages are not simple comments attached to features but rather have to be considered as guidance provided to the user that is based on the current state of the configuration. For instance, in the NXP case, if the selected video codec consumes most of the CPU resources, the configurator should issue a warning advising the user to select another CPU or select a codec that is less resource-demanding. Since they are a needed input for a configurator, they argued that a corresponding construct should exist in TVL.

Ease and cost [C6-C8]. The OSL participant reports that improvements in terms of *model creation* are, in his word, very impressive compared to the graphical notation that was used initially [3]. And since TVL is formally defined, he does not foresee major obstacles to its translation into other formalisms.

The NXP and VirageLogic participants report that, no matter how good the language is, the process of model creation is intrinsically very complex. This means that the cost of model creation is always high for real models. Nevertheless, they observed that the mechanisms offered by TVL facilitate the transition from variability elicitation to a formal specification, hence the neutral score.

Learning experience [C9]. All the participants agreed that the learning curve for software engineers with a good knowledge of programming languages was rather gentle. Software engineers who use only graphical models might need a little more time to feel comfortable with the syntax. In fact, the NXP and VirageLogic participants believe that people in their teams well versed in programming languages would give a ✚ whereas those used to modelling languages would give a ◯, hence their average + score.

6 Findings

This sections builds upon the analysis by looking at the results from three different perspectives: (1) the constructs missed by TVL, (2) the impact of stakeholder profiles on the use of TVL, and (3) the tool support that is provided and still to provide.

6.1 Language constructs

The analysis revealed that extensions to the catalogue of constructs provided by TVL would be appreciated by the participants. We summarise below those that neither make the language more complex nor introduce extraneous information (such as behavioural specifications) into the language.

Attribute cardinality Traditionally, the cardinality of an attribute is assumed to be $\langle 1..1 \rangle$ (`oneOf`), i.e. one and only one element in its domain can be selected. However, the translation of select lists in PloneMeeting would have required enumerations with cardinalities. For instance, in the example in Figure 3, the vote encoders would typically be encoded by a select list, which should be translated into an enumeration with cardinality $\langle 1..2 \rangle$. Even though enumerations are supported by TVL, the absence of cardinality for enumerations forced us to translate them has features. Yet, these select lists typically allow multiple selections, i.e. they require a cardinality like $\langle 1..n \rangle$ (`someOf`). Additionally, optional attributes, like the binding margin, would require a $\langle 0..1 \rangle$ (`opt`) cardinality. Technically speaking, arbitrary cardinalities for attributes are a simple extension of the decomposition operators defined for features. Their addition to TVL will thus be straightforward.

Cloning All the participants expressed a need for cloning in FMs. They have not been introduced in TVL because of the theoretical problem they yield, *viz.* reasoning about potentially infinite configuration spaces and managing clone-specific constraints. Feature cardinalities will thus be proposed with TVL when all the reasoning issues implied by cloning will be solved. This is work in progress.

Default values The main advantage of default values is to speed up product configuration by pre-defining values (e.g. the default page orientation). The participant from OSL argued that, if their applications were to include a TVL-based configuration engine, then TVL should contain default values. This would avoid having default values translated and scattered in the source code, thereby limiting the maintenance effort.

Extended type set Far more types are needed in PloneMeeting than TVL offers. Our experience and discussions with the developers show that only some of them should be built in the language. For this reason, only the `String`, `Date` and `File` types will be added to TVL.

Import mechanism In addition to the `include` mechanism, the participants requested a more fine grained import mechanism. This will require to add scoping to TVL.

Labels and messages In the PloneMeeting case, the participant showed that labels have to be attached to features to provide the user of the configurator with human-readable feature names and description fields and also to propose multilingual content.

Both the NXP and VirageLogic participants use messages to guide the configuration of their products. Constraints on features condition the messages that are displayed based on the current configuration status.

Specialisation In two cases, specialisation mechanisms appeared to be desirable constructs. They would typically allow to overload or override an existing FM, for example, by adding or removing elements from previously defined `enum`, `struct` and feature groups, or by refining cardinalities, e.g. from $\langle 0..* \rangle$ to $\langle 2..3 \rangle$. Although purely syntactical, specialisation mechanisms should facilitate reuse of variability patterns. Note that specialisation differs from references as proposed by Czarnecki *et al.* [3] whose original purpose is *"to modularise a large feature diagram over different diagrams"*.

The extensions that could have been added to TVL but that, we thought, would make it deviate too much from its purpose are the following.

Abstraction Although particularly useful for programming languages, we do not see the interest of using abstract FMs. FMs are, by definition, not meant to be instantiated but to be configured.

Method definition TVL provides a static description of the variability of a PL. Methods, even declarative, would embed behavioural specifications within the language. This is neither an intended purpose nor a desired property of the language. Any method manipulating the FM should be defined externally. [6]

Type parameters In PloneMeeting, type parameters are extensively used. However many of these parameters are very technical and case specific, e.g. editable or searchable fields. Unless we come across more use cases, we will not add parameters to attributes. For cases like PloneMeeting, these parameters can be encoded with the `data` construct.

[6] One may argue that the aggregation functions on attributes are behavioural specifications. In our opinion, they are only syntactic sugar added to ease constraint specification.

6.2 Stakeholder profiles

As shown in Table 2, the participants had fairly different profiles. Our population consists of two developers, one designer and two project managers. Their experience with PLs and FMs also differ. Two participants are intensive users, one is a regular and the other two are still in the transition phase, i.e. moving from traditional to PL engineering.

Interestingly, these profiles did not have a noticeable influence on the marks given to the notation (C1-C3), ease and cost (C6-C8), and learning experience (C9). They all preferred and attribute grammar-like syntax to a markup-based language like XML, usually considered too verbose, unreadable and tricky to edit. Furthermore, the C-like syntax was deemed to preserve many programming habits—like code layout, the development environment, etc.

Deviations appear at the level of modularisation (C4) and expressiveness (C5). One way to interpret it is that OSL and PloneMeeting are still in the transition phase. This means that they are not yet confronted to variability modelling tools in their daily work. They are more familiar with more traditional modelling languages like UML and programming languages like C++ or Python. Compared to these languages, FMs, and TVL in particular, are more specific and thus far less expressive. Furthermore, in the case of PloneMeeting, the participant developed its own all-in-one website configuration and generation tool, which embeds a domain specific language for statechart and class diagram creation.

These observations lead us to conclude that stakeholder profiles:

do no impact the evaluation of the notation. Developers clearly appreciated the textual alternative proposed by TVL. The ease and efficiency of use are the main reasons underlying that preference. Thanks to their knowledge of programming languages, developers naturally prefer to write code than draw lines and boxes. This is usually seen as a time-consuming and clerical task, even with proper tool support. Surprisingly, the participants who were used to graphical models also positively evaluated TVL. They not only liked the language but also the convenience offered by the textual edition interface.

influence the preference for the configuration model. The requirements for the specification of product configurations differed from one profile to another. Developers looked for ways to reuse the TVL model for configuration purposes. Their suggestion was to remove the unnecessary elements from the TVL model and directly use it as a product specification. The advantage of this approach is that comparison between products could be immediately achieved with a simple *diff*. In contrast, the designers were in favour of a graphical model, e.g. a tree-shaped FM, or more elaborate configuration interfaces like configuration wizards or tables.

6.3 Tool support

At the moment, TVL only comes with a parser that checks syntactic and type correctness, as well as a configuration engine that supports decision propagation limited to Boolean values. We have also developed plugins for tree editors, namelly NotePad++ (Windows), Smultron (MacOS) and TextMate (MacOS). These plugins provide basic syntax highlighting and collapse/expand mechanisms to hide/show pieces of code.

Besides textual editors, out-of-the-box versioning tools like CVS or Subversion already support the collaborative editing of TVL models as any other text file, as reported by the OSL and Virage Logic participants. The interpretation of a change made to a TVL line is as easy as it is for programming code. By simply looking at the log, one can immediately see who changed what and when. In contrast, graphical models usually require dedicated tools with their own encoding, which makes interoperability and collaborative work difficult.

The configuration capabilities of TVL have recently been applied to re-engineer the configuration menu of PloneMeeting. This resulted in a prototype that demonstrates how it is possible to use an application-specific web interface as a frontend for a generic TVL-based configurator. Although very limited in functionality, the prototype gave the participant a better overview of the benefits of TVL. Surprisingly, the PloneMeeting participant was not interested in generating *appy.gen* code from a TVL model because of the Python code that would still have to be edited after generation. However, generating a TVL model from *appy.gen* code would greatly simplify constraint specification and validation. Tedious and error-prone Python code would no longer have to be manually maintained, and most of the constraints that are only available in the head of developers would become trivial to implement. Put simply, TVL would be used here as a domain-specific constraint language. We could not produce similar prototypes for the other cases because configuration interfaces were not available and/or access to the code was not granted.

A functionality not provided by the TVL tools but requested by the participants were *code completion* of language constructs, feature names and attributes. Another important functionality would be the *verification of the scope of features* in constraints. Since a constraint can contain any feature in the FM, it might rapidly become hard to identify whether the referenced feature is unique or if a relative path to it has to be given. The on-the-fly suggestion of alternative features by the editor would facilitate constraint definition and make it far less error-prone. By extension, the on-the-fly checking of the satisfiability of the model would avoid wasting time on late debugging. But the latter can lead to resource-intensive computation that should be carefully scheduled and optimized.

7 Threats to Validity

The evaluation was performed with four PLs and five participants, providing a diversity of domains and profiles. Yet, the outcomes of the evaluations were very similar across the cases. Therefore, we believe that the results are valid for wide a range of organizations and products [38].

The TVL models were prepared in advance by the two researchers and later checked by the participants. Consequently, the expertise of the researchers might have influenced the models and the evaluation of the participants. In order to assess this potential bias more precisely, we will have to compare models designed by participants to models designed by the two researchers. However, TVL is arguably simpler than most programming languages and the modelling task was felt to be rather straightforward. As as consequence, we do not expect this to be a problem for our evaluation.

A more specific problem was the unavailability of proper documentation and the limited access granted to the codebase in the case of OSL, NXP and Virage Logic. This made the modelling of the case more difficult.

In the OSL case, the development team is still in the SPL adoption phase. This could be a threat as the participant has only been exposed to FMs for reviewing. Therefore, he might have focused on comparing the textual and graphical approaches rather than evaluating the language itself. Along the same lines, the PloneMeeting participant was already reluctant to use graphical FMs and might have evaluated the textual approach rather than TVL itself.

More generally, one limitation of our study is the relatively small size if the subsystems we could deal with during the experiment.

8 Conclusion

Effective representations of feature models are an open research question. Textual alternatives have been proposed, including TVL, a textual variability modelling language meant to overcome a number of known deficiencies observed in other languages. Yet, evidence of the suitability of TVL in practice was missing. In this paper, we systematically evaluated the language by conducting an empirical evaluation on four industrial product lines.

Our evaluation of TVL showed that practitioners positively evaluated the notation and that efficiency gains are envisioned in terms of model comprehension, design and learning curve. However, they suggested some extensions like attribute cardinalities, feature cloning, default values and guidance messages that can be used during product configuration.

In the future we will focus on integrating the recommended extensions into TVL. Furthermore, the prototype implementation of the TVL parser and reasoner needs to be extended to better support on-the-fly verification of model consistency. To assess these new extensions, live evaluations through modelling sessions are envisaged. To better asses the pros and cons of variability modelling languages, comparative evaluations are planned, too.

Acknowledgements

We would like to thank Gaëtan Delannay from GeezTeem, Jacques Flamand from OSL, Hans den Boer and Jos Hegge from Virage Logic for the time they dedicated to our evaluation, and Bart Caerts from NXP Semiconductors for his contribution to the development of the CPU-load calculation variability model. This work is sponsored by the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy, under the MoVES project, the Walloon Region under the European Regional Development Fund (ERDF) and a FIRST DOC.A Fund.

References

1. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, SEI, Carnegie Mellon University (November 1990)
2. Pohl, K., Bockle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer (July 2005)
3. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice **10**(1) (2005) 7–29
4. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks, doi:10.1016/j.comnet.2006.08.008, special issue on feature interactions in emerging application domains (2006) 38
5. pure-systems GmbH: Variant management with pure::variants. http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf (2006) Technical White Paper.
6. van Deursen, A., Klint, P.: Domain-specific language design requires feature descriptions. Journal of Computing and Information Technology **10** (2002) 2002
7. Batory, D.S.: Feature Models, Grammars, and Propositional Formulas. In: SPLC'05. (2005) 7–20
8. Boucher, Q., Classen, A., Faber, P., Heymans, P.: Introducing TVL, a text-based feature modelling language. In: VaMoS'10, University of Duisburg-Essen (January 2010) 159–162
9. Czarnecki, K.: From feature to concept modeling. In: VaMoS'10, University of Duisburg-Essen (January 2010) 11 Keynote.
10. Nestor, D., O'Malley, L., Sikora, E., Thiel, S.: Visualisation of variability in software product line engineering. In: VaMoS'07. (2007)
11. Cawley, C., Healy, P., Botterweck, G., Thiel, S.: Research tool to support feature configuration in software product lines. In: VaMoS'10, University of Duisburg-Essen (January 2010) 179–182
12. Dordowsky, F., Hipp, W.: Adopting software product line principles to manage software variants in a complex avionics system. In: SPLC'09, San Francisco, CA, USA (2009) 265–274
13. Classen, A., Boucher, Q., Faber, P., Heymans, P.: Syntax and semantics of TVL, a comprehensive text-based feature modelling language. Technical report, PReCISE Research Centre, Univ. of Namur (2009)
14. Shull, F., Singer, J., Sjøberg, D.I.K.: Guide to Advanced Empirical Software Engineering. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007)
15. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Feature Diagrams: A Survey and A Formal Semantics. In: RE'06. (September 2006) 139–148
16. Antkiewicz, M., Czarnecki, K.: Featureplugin: feature modeling plug-in for eclipse. In: OOPSLA'04. (2004) 67–72
17. Beuche, D.: Modeling and building software product lines with pure: :variants. In: SPLC '08: Proceedings of the 2008 12th International Software Product Line Conference, Washington, DC, USA, IEEE Computer Society (2008) 358
18. Kästner, C., Thüm, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., Apel, S.: FeatureIDE: A tool framework for feature-oriented software development. In: Proceedings of ICSE'09. (2009) 311–320
19. Krueger, C.W.: Biglever software gears and the 3-tiered spl methodology. In: OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, New York, NY, USA, ACM (2007) 844–845
20. Mendonca, M., Branco, M., Cowan, D.: S.p.l.o.t. - software product lines online tools. In: Proceedings of OOPSLA'09. (2009) 761–762

21. Mendonça, M.: Efficient Reasoning Techniques for Large Scale Feature Models. PhD thesis, University of Waterloo (2009)
22. Reiser, M.O.: Core concepts of the compositional variability management framework (cvm). Technical report, Technische Universität Berlin (2009)
23. Abele, A., Johansson, R., Lo, H., Papadopoulos, Y., Reiser, M.O., Servat, D., Torngren, M., Weber, M.: The cvm framework - a prototype tool for compositional variability management. In: Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), University of Duisburg-Essen (January 2010) 101–105
24. Benavides, D., Segura, S., Trinidad, P., Cortés, A.R.: Fama: Tooling a framework for the automated analysis of feature models. In: Proceedings of VaMoS'07. (2007) 129–134
25. Riebisch, M., Böllert, K., Streitferdt, D., Philippow, I.: Extending feature diagrams with uml multiplicities. In: IDPT'02. (2002)
26. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.W.: Generative programming for embedded software: An industrial experience report. In: GPCE'02, London, UK, Springer-Verlag (2002) 156–172
27. Pratt, T.W.: Programming Languages : Design and Implementation. Second edition edn. Prentice Hall (1984) 604 pages.
28. Holtz, N., Rasdorf, W.: An evaluation of programming languages and language features for engineering software development. Engineering with Computers **3** (1988) 183–199
29. Moody, D.L.: The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. IEEE Transactions on Software Engineering **35** (2009) 756–779
30. PloneGov. http://www.plonegov.org/ (June 2010)
31. Delannay, G., Mens, K., Heymans, P., Schobbens, P.Y., Zeippen, J.M.: Plonegov as an open source product line. In: OSSPL'07, collocated with SPLC'07. (2007)
32. Hubaux, A., Heymans, P., Benavides, D.: Variability modelling challenges from the trenches of an open source product line re-engineering project. In: SPLC'08, Limerick, Ireland (2008) 55–64
33. Unphon, H., Dittrich, Y., Hubaux, A.: Taking care of cooperation when evolving socially embedded systems: The plonemeeting case. In: CHASE'09, collocated with ICSE'09. (May 2009)
34. Laboratories, O.S. http://www.osl.be/ (June 2010)
35. NXP Semiconductors. http://www.nxp.com/ (June 2010)
36. Virage Logic. http://www.viragelogic.com/ (June 2010)
37. Sangwan, S., Hian, C.K.: User-centered design: marketing implications from initial experience in technology supported products. In Press, I.C.S., ed.: Engineering Management Conference. Volume 3. (2004) 1042– 046
38. Yin, R.K.: Case Study Research: Design and Methods. 3rd edn. Volume 5 of Applied Social Research Methods. Sage Publications, Inc (December 2002)