Variability Modelling Challenges from the Trenches of an Open Source Product Line Re-Engineering Project

A. Hubaux, P. Heymans
PReCISE Research Centre
Faculty of Computer Science
University of Namur
{ahu, phe}@info.fundp.ac.be

D. Benavides
Department of Computer
Languages and Systems
University of Seville
benavides@us.es

Abstract

Variability models, feature diagrams ahead, have become commonplace in the software product lines engineering literature. Whereas ongoing research keeps improving their expressiveness, formalisation and automation, more experience reports on their usage in real projects are needed. This paper describes some challenges encountered during the re-engineering of PloneMeeting, an Open Source software family, into a software product line. The main challenging issues we could observe were (i) the ambiguity originating from implicit information (missing definitions of feature labels and unclear modelling viewpoint), (ii) the necessity of representing spurious features, (iii) the difficulty of making diagrams and constraints resistant to change, and (iv) the risks of using feature attributes to represent large sets of subfeatures. Our study reveals the limitations of current constructs, and calls for both language and methodological improvements. It also suggests further comparative evaluations of modelling alternatives.

1 Introduction

Feature diagram (FD) languages are a common means to represent variability in Software Product Lines (SPL). FDs come in various flavours, a large part of which have been surveyed and formalized in [20]. FD languages have also been compared *theoretically*, according to mathematical criteria [21, 11].

These results can be used by method engineers and language designers for choosing the constructs to be included in their custom FD languages, and for formalizing them quickly. The results are also valuable to tool developers for ensuring the correctness and efficiency of their reasoning algorithms.

However, such criteria and results can certainly not answer all questions about the appropriateness of current feature modelling techniques. For instance, although formalized, expressively complete and having a host of efficient algorithms at hand, a language might well turn out to be unsuitable for a given modelling purpose. This can happen, e.g., because the language lacks appropriate constructs to convey the intended meanings, because the diagrams quickly become unreadable, or because insufficient methodological guidance is provided to the modeller [15].

These kinds of issues can only be answered *empirically*, by confronting the modelling techniques to real usage situations. This is what the present paper has to offer, thereby complementing the previous investigations. This research appeared to us as a necessity also because, despite the availability of experience reports dealing with variability management in SPL (e.g., [23]), we found no thorough empirical study focusing on the assessment of feature modelling techniques. A possible reason for this is the risk of disclosing strategic corporate information potentially contained in actual FDs.

This situation drove us to look for a significant and open case study allowing free dissemination of results in the research community. These requirements were met by PloneGov, an open source project developing *eGovernment* applications [10] based on Plone [5]. More precisely, we focus on PloneMeeting, a Plone-based family of products intended to support the management of meetings held by local and regional authorities. The current activity in which we and PloneMeeting developers cooperate is the reengineering of PloneMeeting's configuration subsystem. In pursuing this endeavour, it quickly became apparent that both the developers and the researchers were missing a clear and agreed view of the variability currently supported by the existing software, what we call *software variability* (as opposed to *product line (PL) variability* [16]). We thus started to elicit the software variability from the existing artefacts, and reported some preliminary observations in [12]. In the

978-0-7695-3303-2/08 $25.00 © 2008 IEEE
DOI 10.1109/SPLC.2008.39

present paper, we focus on a set of challenges encountered when applying FDs to model the previously elicited information.

The rest of this paper is organised as follows. Sec. 2 will recall the notion of software variability and the FD language that we used. In Sec. 3, we will introduce PloneGov, PloneMeeting and the settings of our study. The four following sections are each devoted to a challenge. Sec. 4 reveals the ambiguity arising from implicit information (missing definitions of feature labels and unclear modelling viewpoint). Sec. 5 shows the necessity and embarrassment of representing spurious features. Sec. 6 emphasises the difficulty of making diagrams and constraints resistant to change. Sec. 7 highlights the risks of using feature attributes to represent large sets of subfeatures. After the detailed account of each challenge, Sec. 8 will summarize our contributions and broaden the discussion by addressing the threats to validity. Sec. 9 will wrap-up the paper and announce our future work.

2 Modelling Software Variability with Feature Diagrams

2.1 Software Variability

In the reported study, our goal is to model the *software variability* of PloneMeeting. We do so to obtain a clear and agreed picture of the capabilities of the existing assets before further ado. As advocated in our previous work [16], we consider essential to distinguish between software and PL variability.

Software variability refers to the *"ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context"* [22]. This kind of variability is well known from the development of single systems, and is thus not specific to SPL. As an example, an abstract Java super-class allows different specializations to be used where the super-class is used.

PL variability [6, 19, 14], on the other hand, is specific to SPL and describes the allowed variation between the systems that belong to a PL. Defining PL variability is an explicit decision of product management [14, 19]. As an example, product management might have decided that the mobile phones of their PL should either offer the GSM or the UMTS protocol, but not both (although the existing assets might well technically allow to have both at the same time, which would be software variability).

In [16], the authors proposed to record PL and software variability in separate models (e.g., FDs) and to interrelate them. Equiped with formal syntax and semantics, those models not only disambiguate the documentation of variability, but are also amenable to automated analyses. In the present paper though, we only focus on the challenges related to the modelling of the software variability of PloneMeeting. Confronting PloneMeeting's PL and software variabilities is part of our future work.

2.2 The FD language of Czarnecki *et al.* [8]

FDs are a common means to model variability, be it PL or software variability. For our study, we resorted to the cardinality-based FD language proposed by Czarnecki *et al.* [8]. This language was chosen because it was deemed representative of the state of the art in FD languages: being a popular notation, being expressively complete [21], using cardinality-based decomposition operators (which have been shown to subsume the other traditional operators *or*, *and*, *xor*...[21]), possessing a formal semantics as well as advanced constructs like feature attributes, feature cloning and references. It is important to emphasise that we do not consider this paper as an assessment of this language in particular, but rather as an evaluation of the current state of the art in feature modelling.

We assume familiarity of the reader with the modelling language. Here, we simply recall some notational conventions and terminology by referring to an example further elaborated in the rest of the paper. *Group cardinality* (often simply called cardinality afterwards) is represented by an interval of integers between the symbols '<' and '>'. An example is `<1..1>` appearing thrice in Fig. 5. Group cardinality indicates the minimum and maximum numbers of features, from a specific group of subfeatures, that can appear in one configuration of the FD, given that their parent feature is part of the configuration too. It should not be confused with *feature cardinality*, e.g. `[1..*]`, which indicates how many times the same feature can appear in one configuration given that its parent feature appears there too. In this paper, we happen to only use default values of feature cardinality: `[0..1]` is depicted by a hollow circle on top of the feature, therefore called *optional* feature (e.g., *Fallback* in Fig. 5); `[1..1]` is depicted by a filled circle topping the feature, thus called *mandatory* (e.g., *Translations* in Fig. 5). Other constructs will be introduced when needed.

3 The PloneMeeting Case Study

PloneMeeting is part of PloneGov, an international Open Source (OS) initiative promoting secure, collaborative, and evolutive *eGovernment* web applications. PloneGov fosters cooperative development of such applications to benefit from economies of scale. PloneMeeting was initiated by the Belgian government to offer advanced meeting management support to national (local and regional) authorities. However, it has also been gaining attention from beyond

the Belgian borders, and is currently being tested in some French, Spanish and North American towns. The following subsections will introduce PloneGov, PloneMeeting as well as our research objectives and method within this context.

3.1 The PloneGov initiative

PloneGov [1] is based on Zope and Plone [5]. Zope is an OS application server for hosting content management systems (CMS), intranets, portals, and custom applications. Plone is a portal and CMS built on top of Zope.

PloneGov gathers around 55 European, American and African public organizations. It consists of 17 projects with more than 10 product releases [1]. Products are divided into (1) citizen-oriented services, (2) government internal applications, of which PloneMeeting, and (3) general purpose tools.

The worldwide scope of PloneGov yields many contexts to deal with. Those differ in specific legal, social, political and linguistic characteristics than influence the features required for a given product.

3.2 The PloneMeeting product family

Although PloneMeeting would be called a *product* in Plone jargon, PloneMeeting is actually a *family* of products (see Fig. 1) supporting the organisation, performance and follow-up of official meetings held by local and regional authorities. PloneMeeting is built on top of Plone and thus reuses Plone software assets extensively. Its development started in June 2007 and a working version is now available on the PloneGov website[1]. Similarly to PloneGov, PloneMeeting products differ in characteristics determined by the context of use. For example, the display language (Dutch, French, English...), the meeting management workflows, the recognized document formats, the GUI's look and feel... must adapt to local constraints.

Two sub-families of the PloneMeeting family – called *profiles* in Plone – are PloneMeetingCommunes[2] and EGW. These profiles need to be further refined into *configurations* – another Plone term – to allow the full specification of a running instance, i.e., a family member. Fig. 1 sketches this situation and suggests that variation points are likely to be present at different levels, suggesting a non trivial variability management and resolution process. The red rectangle on the left-hand side gives an overview of the distribution of stakeholders according to the targeted variability resolution stage. Four main classes of stakeholders were identified. The *developers* are those in charge of designing and producing the application code. We distinguish Plone and PloneMeeting developers. The *system administrators* are those in charge of configuring and maintaining the PloneMeeting websites, guided by *domain experts*. *Users* are the civil servants using the websites to manage meetings.

There are three different products (a.k.a. family members, a.k.a. Plone configurations) currently available in PloneMeeting: *Communal Council*'s goal is to handle meetings of the communal council; *Communal College* deals with the meetings of the communal college; *e-gw* focuses on the meetings of the Walloon government[3].

3.3 Research question and method

The research question addressed in the reported case study can be formulated as: *What are the practical obstacles encountered when applying state-of-the-art FD languages to the modelling of software variability of a real size product line?*

Considering the openness of the question, the fact that it is insufficiently addressed by current research, and since answering it imposes to operate in complex settings where there is little control over variables, an *exploratory case study* [18] appeared to be an appropriate approach.

As mentioned in the introduction, our study takes place within a larger project in which we are assisting the PloneMeeting developers in re-engineering their configuration system. Currently, PloneMeeting is configured through a mix of many manual and software-assisted tasks performed at various application levels (Zope, Plone, PloneMeeting, browser...), by different kinds of stakeholders and at different times. This process is judged too complex, too time-consuming and error-prone (it is easy to make inconsistent configuration choices). A recognized source of the problem was that there is no overall view of the variability of the application among the developers, which could be exploited to generate a more reliable configurator. The task of inventorizing the software variability that the software assets are currently capable of, has thus been given high priority by the developers. This task involves (1) eliciting the information and (2) documenting (i.e. modelling) it. The latter subtask is the focus of the present paper. The former is reported in [12] and quickly summarized in the next paragraph.

The *elicitation* task started with meetings involving the PloneMeeting developers and us, the researchers. We then attended a demo session of the Belgian PloneGov products from which we extracted a broad overview of the PloneGov initiative. Attending the coding sessions allowed us to better identify the different products that the PloneMeeting developers are working on, and to extract some detailed variability information. Extraction of this information was complemented by the analysis of several PloneMeeting configuration menus and files, as well as the project's *trac* [4].

[1] `http://www.plonegov.org/products/plonemeeting`

[2] A commune is a Belgian local (city or town) authority.

[3] Wallonia is one of the three territorial regions of Belgium.

[4] A trac is a web-based software project management and bug/issue

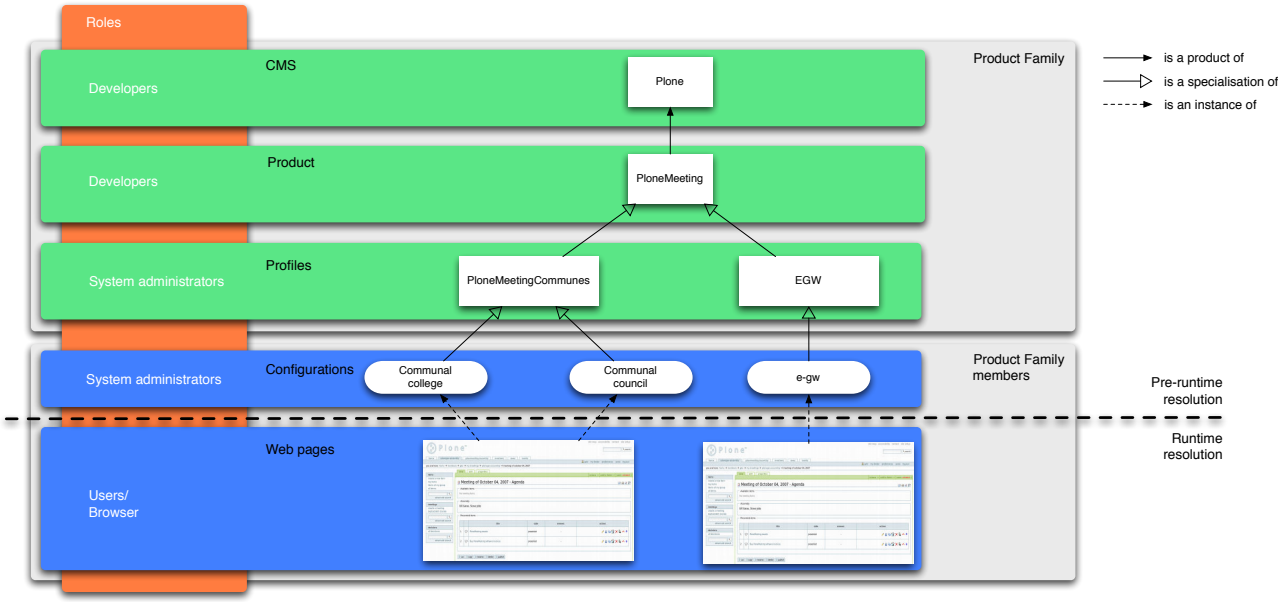


Figure 1. Overview of PloneMeeting variability resolution levels

The elicitation requested an effort of one man-month, performed by a junior researcher (the first author) between October and December 2007.

The *modelling* task started four weeks after the start of the elicitation phase but largely overlapped with it. The effort was of about one man-month. The mission of the researcher was to use the language introduced in Sec. 2.2 in order to represent faithfully the elicited information. In order to avoid a tool bias, there were no specific tools imposed, just a generic diagramming tool (OmniGraffle). Again, the performer of this task was the first author, who kept track systematically of all the problems he encountered during the modelling.

Several recurring challenges quickly surfaced. In the following sections, we characterise these challenges in detail. Each of them is illustrated through a representative situation in which it occurred: we will first explain the situation to be modelled independently from the modelling notation; secondly, we will describe the tentative model(s) made with FDs, the problems we ran into and discuss the pros and cons of the various modelling alternatives; finally, we will draw the lessons and emphasize open research issues. The order in which the challenges appear in the text is neither representative of their importance, nor of the sequence in which we encountered them. They are presented so as to facilitate comprehension. However, only the challenges that were deemed the most important in this initial modelling phase are reported. Other challenges, some of which might gain importance as the case study progresses, are briefly mentioned in sec. 8. There, we also examine the threats to the validity.

tracking system. The PloneMeeting trac is available at `http://dev.communesplone.org/trac/report/120`

4 Challenge 1: implicit modelling viewpoint

4.1 Situation to be modelled

The availability of Plone in many languages is dealt with by the Plone internationalisation (*i18n*) initiative, a result of which is Plone's built-in translation management service, the so-called PlacelessTranslationService (PTS) [5]. It provides a dynamic translation mechanism independent from string locations in web pages. Translations are stored in key/value pairs. A key is a label in the code identifying a translatable string; the value is its translation.

The developers are in charge of providing the translation files. The system administrators and the users are those able to select the language to be used. Physically speaking, all the Plone translation files are copied at each installation, or website instance, as shown in Fig. 2. However, at runtime, the file actually in use may vary. The files appearing with a grey background in Fig. 2, viz. `Plone-en.po` in installation 1 and `Plone-fr.po` in installation 2, are those used at a given time t of the execution.

4.2 Issues faced when applying FDs

When starting to model the software variability of the PTS, we first drew a diagram such as the one in Fig. 3(a),

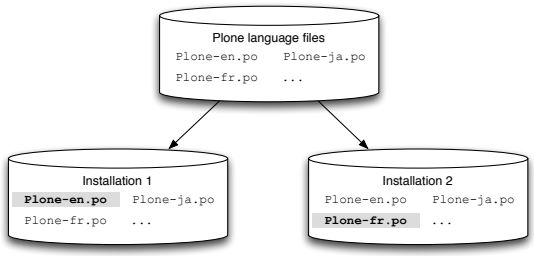


Figure 2. Presence and usage of files in two Plone installations at execution time t

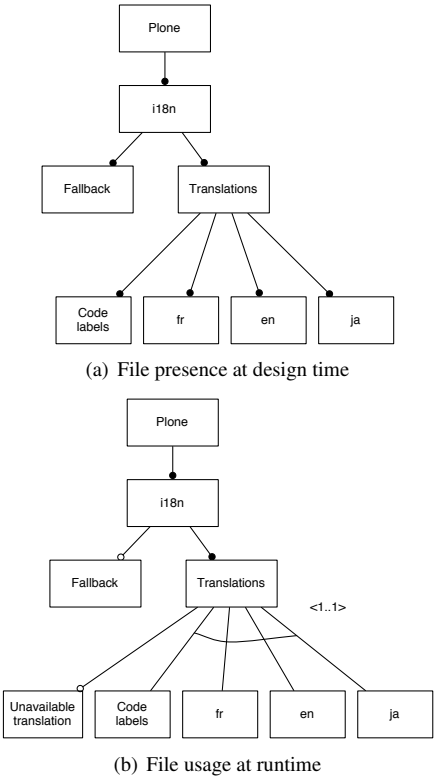
intuitively interpreting features *fr*, *en*, *ja*[5] as denoting the *presence* of files `Plone-fr.po`, `Plone-en.po` and `Plone-ja.po`, respectively. (The reader should ignore features *Fallback* and *Code Labels* for the moment). However, as it appears, Fig. 3(a) contains no variability (all features are mandatory) and therefore fails to represent important information that was elicited: the fact that only one file can be used at a time[6].

As a consequence, we drew a second diagram, such as the one in Fig. 3(b). (The feature *Unavailable Translation* should be ignored for the moment). This one interprets features *fr*, *en*, *ja* as denoting the *usage* of files at a given time and constrains them with a feature group cardinality of `<1..1>`, faithfully representing the elicited information.

So, although it took a second try, the modelling of the situation was no problem in the end. However, the *a posteriori* interpretation of the diagrams by the developers appeared to be a problem, as there was no indication of the way to interpret the features in the diagram. This might turn out to be a very annoying issue when we consider that the developers will have to build a configurator that implements, or interprets, the constraints found in the diagram. Here, without the indication of the meaning of the features, there is no way for the developers, or the interpreter, to determine the time at which the resolution of variability must take place, that is, the *binding time* [22].

4.3 Lessons learned and perspectives

A first lesson that we draw from this challenge is that an explicit indication of the modelling perspective, or viewpoint, (here, presence *vs.* usage of files) might be a precious indication for interpreting an FD. In this case, the perspectives happened to correspond to two different binding times (design time and runtime, respectively), but other perspectives might be possible. Binding time has been discussed as a modelling perspective in [7].



(a) File presence at design time

(b) File usage at runtime

Figure 3. Plone *i18n* variability models

In addition to the *a posteriori* indication of the perspective, it would also be interesting to have some sort of *a priori* list of recurring perspectives that it might be worth to consider, acting as a checklist for the elicitation and modelling activities.

Another methodological suggestion is to provide definitions for all feature labels used in FDs. This recommendation, although already advocated in feature modelling [13] and other modelling domains [25], is often overlooked in practice. Maybe a lighter, or complementary suggestion, relevant when dealing with software variability, is to use typed traceability links between the features and the assets that they are about. In our case, link types such as `<possesses>` (for indicating presence) or `<uses>` (for usage) could help disambiguate the meaning of the features wrt the files they point to.

Finally, we should note that solutions have been suggested for the specific issue of discriminating between different binding times within the same FD [13, 24]. Surprisingly, we did not encounter a situation requiring this in our case study (yet), but we could well imagine a runtime variability that only becomes relevant depending on some design time choice. This might be the case, for example, if

[5] Here, we assume a small set of languages (French, English and Japanese) for simplicity.

[6] This constraint is actually not true in all contexts of use of Plone, but it is in the one we considered.

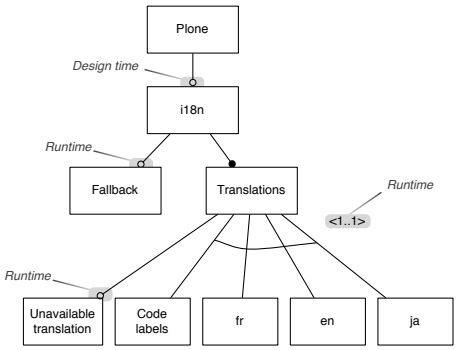


Figure 4. Plone *i18n* FD: design and runtime together

i18n was a design time option. In such a situation, an FD where each variation point is tagged with a binding time (see Fig. 4) would become valuable. However, the binding time construct currently possesses no formal semantics, and thus calls for an adequate formalization that will substantially change the one presented in [20]. We also encourage a more accurate comparison of the pros and cons of two approaches: putting different binding times in different diagrams *vs.* grouping them in the same diagram.

5 Challenge 2: "unavailable" features

5.1 Situation to be modelled

The so far eluded Plone *i18n* "fallback" mechanism [5] posed some modelling problems too. To understand how fallback works, we need to clarify the way the display language of a running Plone instance is selected. The language is actually determined by the PTS according to the language of the web browser. For instance, if the browser is set to *fr* (French), the PTS will choose *fr* as display language. However, since the PTS only knows a limited set of languages, the browser language might well not be in the PTS's list. This is where fallback intervenes: (c_1) if fallback is enabled, and the selected language is unavailable, e.g. *wal* (Walloon), use the default, typically *en* (English); (c_2) if fallback is enabled, and the selected language is available, e.g. *fr*, use it; (c_3) if fallback is disabled, and the selected language is available, use it; and (c_4) if fallback is disabled, and the selected language is unavailable, use code labels. Fallback can be enabled or disabled by the system administrator via the Zope Management Interface (ZMI)[7].

[7]The ZMI is a management and development environment through which Zope is controled, Zope objects are manipulated, and web applications are developed.

5.2 Issues faced when applying FDs

To model the above situation, we started from the FD in Fig. 3(b) which, at the time, did not have features *Fallback*, *Code labels* and *Unavailable translation*, yet. We first added *Fallback*, an optional runtime feature, and *Code labels*. The fact that only one language can be displayed at a time is easily modelled by the `<1..1>` group cardinality. However, for representing constraints c_1-c_4, we had to resort to other means.

We first tried with graphical crosscutting constraints (`<requires>` and `<excludes>`) and their textual equivalent, Composition Rules (CR) [13]. However, the diagrams (resp. formulae) quickly became cluttered and, most importantly, were insufficient to convey the meaning of the constraints. This was no surprise, as the expressive weakness of the aforementioned constraint languages has already been pointed out in the literature [3, 21].

Following [3], we thus opted for propositional logic formulae. This turned out to meet our modelling needs, but required the addition of a somewhat spurious feature, *Unavailable translation* (*UT*, for short), see Fig. 3(b). Since we could not realistically imagine to have a feature for each unavailable language, the tentative choice of such a language by the browser was represented by *UT*. Then, we could add the following constraints:

$$\begin{array}{ll} (\phi_1) & UT \wedge Fallback \Rightarrow en \\ (\phi_2) & UT \wedge \neg Fallback \Rightarrow Code\ labels \\ (\phi_3) & (fr \vee ja) \Rightarrow \neg UT \end{array}$$

ϕ_1 accurately represents c_1, ϕ_2 represents c_4, and ϕ_3 covers both c_2 and c_3.

5.3 Lessons learned and perspectives

A first lesson drawn from this example is the confirmation that crosscutting constraints and CR are not sufficient to model some real situations accurately.

Most important is the observation that although a more powerful constraint language, such as propositional logic, can overcome this issue, it still forces the modeller to harm the "naturalness" of the FD by adding a spurious feature. We call this feature spurious because it represents a set of features (translation languages) that are actually *not* offered by current software assets, which is arguably paradoxical when modelling the provided software variability. This also reinforces the importance of carefully specifying the meaning of features in definitions accompanying the FD (see Sec. 4). In some cases, it might also be concluded that there is no way to model the situation in a reasonably natural way based on FDs. For example, when dynamic aspects are too complex, it might be more appropriate to turn to behavioural modelling techniques, such as UML State Diagrams.

Situations like this one appeared in several other parts of our case study, suggesting the emergence of some sort of pattern. We think that it would be useful to document such patterns and the way to model them in methodological guidelines for FD modelling.

6 Challenge 3: evolving variability

6.1 Situation to be modelled

The present challenge originated from the need to update previously modelled variability information. This happened, for example, as new translation files were made available. Therefore, the information contained in the FD of Fig. 3(b) and the documentation of constraints c_1-c_4 had to be adapted. As these kinds of changes actually take place regularly (especially in OS projects where new assets are continuously uploaded and local installations updated) models must be easily evolvable. In our case, we looked for a resilient modelling of the runtime fallback mechanism.

6.2 Issues faced when applying FDs

We started from the FD of Fig. 3(b) and its side constraints ϕ_1, ϕ_2 and ϕ_3. Updating the FD was relatively straightforward, so we do not discuss it here (but return to this issue in Sec.7). For now, we only focus on the constraints. If *en* and *Code labels* remain the default choices (resp. with fallback enabled and disabled), ϕ_1 and ϕ_2 do not have to change. However, ϕ_3 has to. For example, if Spanish (*es*) and Italian (*it*) are added, it becomes $(fr \vee ja \vee es \vee it) \Rightarrow \neg UT$.

Manual change of this formula (and possibly others) each time a new translation file is added (or removed), is a cumbersome process. It increases the risk of inconsistency between the FD and the side constraints, and among the constraints, especially if there are many translation languages. So, we looked for a way to remove ϕ_3, or make its formulation more evolvable.

We could make ϕ_3 disappear through a refactoring of Fig. 3(b) into Fig. 5. The latter FD adds two intermediate features under $Translations$, namely $DefaultTranslations$ (DT) and $NonDefaultTranslations$ (NDT). Mutual exclusion between the translation languages is now guaranteed by three (instead of one) `<1..1>` cardinalities, and although ϕ_1 and ϕ_2 remain the same, ϕ_3 is now replaced by an `<excludes>` link between UT and NDT.

Alternatively, it appeared possible to keep Fig. 3(b) as is, and only rewrite ϕ_3 so that it can be changed more easily, at least with respect to the addition of non default languages:

$$(\phi'_3) \quad (Translations \wedge \neg(en \vee Code\ labels)) \Rightarrow \neg UT$$

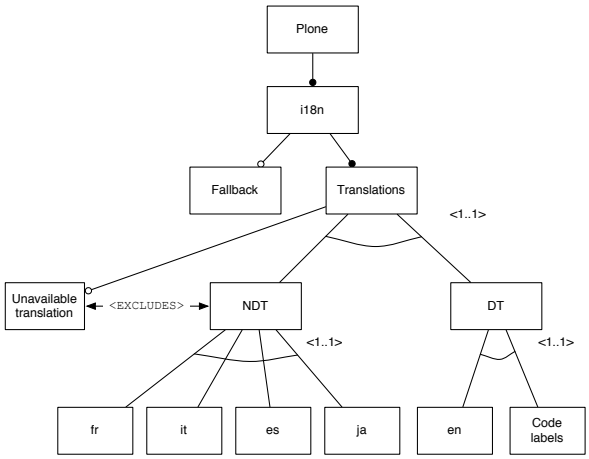


Figure 5. Refactored Plone *i18n* runtime FD

6.3 Lessons learned and perspectives

The first solution (the refactoring of the FD) reveals the existence of a trade-off between the simplicity (in size and structure) of an FD, on the one hand, and its evolvability on the other. This is an issue that certainly deserves further investigation, which, in the end, should result in modelling guidelines indicating when it is worth complicating the FD and when it is not.

The second solution (rewriting the formulae) seems simpler and less costly, but we need further empirical evidence to check if it will always be the best choice.

In doubt, we also envisaged an alternative solution in the form of language enhancements that would make model refactorings such as the above unnecessary. The suggestion is to allow gathering features of the FD into "logical sets", irrespective of where they appear in the FD (thereby not affecting the decomposition structure). If such sets were given names, and if we extend propositional formulae with some kind of macro constructs, we might end up with formulae that are more resistant to continuous variability evolution.

In our example, leaving Fig. 3(b) as is, we could name NDT the set of non default language translation features, and define it as: $NDT = \{fr, ja, es, it\}$. Then, we could replace ϕ_3 or ϕ'_3 with a formula like

$$(\phi''_3) \quad \bigvee_{f \in NDT} f \Rightarrow \neg UT$$

where $\bigvee$ is a macro whose expansion concatenates $\vee$ operators and their operands.

This proposal is by no means a complete nor definitive solution. Firstly, the macro-enhanced constraint language would have to be defined precisely. Secondly, the logical sets' definitions would still have to be maintained manually.

This is however probably less complex and error-prone than directly maintaining all the concerned constraints, or refactoring the FD. Again, a more thorough investigation of the pros and cons of the modelling alternatives is an interesting topic for future research. The use of OCL path expressions, as suggested in [9], is also a possible alternative.

7 Challenge 4: large sets of subfeatures

7.1 Situation to be modelled

In the previous descriptions of *i18n*, the number of languages has been voluntarily restricted for readability purposes. However, Plone actually generates content in more than 50 languages[8].

7.2 Issues faced when applying FDs

Given the simplified settings (3 languages), we could afford to model all languages extensively in Fig. 3(b): one language becomes one feature. However, for 50+ languages, we considered replacing this option (see Fig. 6(a)) by other alternatives (Fig. 6(b) and Fig. 6(c)).

Both alternatives follow a suggestion found in [8] and [4] to replace subfeatures by attribute values of the superfeature, in this case $Translations$. The type of the attribute appears between parentheses below the superfeature's name. This allowed to reduce the number of subfeatures to one, thereby significantly downsizing the FD. However, by using an attribute of type $String$ and not further constraining it, the first alternative (Fig. 6(b)) was not accurate wrt the modelled situation: an explicit list of the supported languages was deemed an important information on the current software capabilities; it should be documented at this stage to avoid forgetting about it later.

With the second alternative (Fig. 6(c)), we solved this issue by using an enumerated type, here called $Language$. In this case, the informational content is the same as in Fig. 6(a). Nevertheless, some adaptation of the constraints ϕ_1, ϕ_2 and ϕ'_3 was needed (Tr stands for $Translations$):

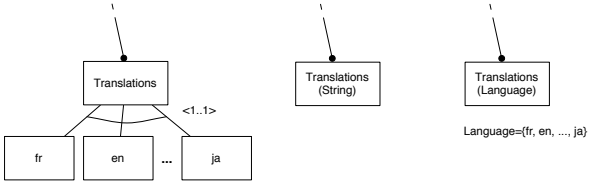
$$(\phi_1) \quad UT \wedge Fallback \Rightarrow Tr = en$$
$$(\phi_2) \quad UT \wedge \neg Fallback \Rightarrow Tr = Code\ labels$$
$$(\phi'_3) \quad (Tr \neq en \wedge Tr \neq Code\ labels) \Rightarrow \neg UT$$

7.3 Lessons learned and perspectives

A first observation is that although feature attributes definitely allow to reduce the size of FDs, resorting to them too hastily (e.g., using "open" types like $String$) could easily lead to loss of information with potentially harmful consequences in later development stages.

Reconsidering evolvability (the previous challenge), we could also imagine a solution that combines arbitrary "logical" (i.e., decomposition-independent) feature sets (see Sec. 6.3) with typed feature attributes using type constructors such as $\cup$, $\cap$, $\times \ldots$ For example, in our case, we could define $Language = DT \cup NDT$ with NDT and DT in turn defined as $NDT = \{fr, ja, es, it, \ldots\}$ and $DT = \{en, Code\ labels\}$. The formalisation and applicability of this suggestion remains a topic for future work, but we can already observe that it would combine evolvability and concision, the latter remaining a concern in Fig. 5.

Finally, we should also keep in mind that attributes can only replace terminal (leaf) features. Different or adapted solutions, e.g., using FD references [9], should be investigated for large sets of subfeatures that include non terminal ones.



(a) Features (b) Attribute of generic type (c) Attribute of enumerated type

Figure 6. Modelling many subfeatures in FDs

8 Discussion

Based on the issues encountered during the modelling of PloneMeeting's software variability, the previous sections called for a number of language enhancements, methodological guidelines and more thorough comparative evaluations of modelling alternatives. However, the relevance of those research directions to the field of feature modelling in general has to be confronted with the peculiarities of our study. As recommended by best practice in case study research [18], we thus now discuss the threats to validity.

The relative lack of experience of the appointed researcher is certainly such a threat. At the start of the study, he was a first year PhD student, with a solid background in Software Engineering, but no particular experience in feature modelling. So, firstly, as experience can compensate for the lack of explicit methodological guidelines, one could argue that the latter issue is overrated here. However, we believe that a guide of best practice for feature modelling would be a valuable asset for practitioners since (1) newcomers arrive continually and (2) such a guide is likely to

[8]Whose list is available at `http://plone.org/development/teams/i18n/existing-translations/?searchterm=plone\%20available\%20translation`

harmonize modelling styles among experienced modellers too, thereby preventing interpretation errors. Secondly, the lack of experience of the researcher might have lead us to focus on issues that an experienced modeller would have solved seamlessly. This possible bias might be reinforced by the relatively short time (1 man-month) spent on the case study. The modelling of PloneMeeting's variability is not over, and further work might reveal new challenges or lead us to reconsider the importance given to the previous ones.

For instance, a challenge we only touched at, and whose importance is likely to grow as we progress, is model scalability. As we have seen, a construct like feature attributes, or others like feature cloning or references [8], help master the size of the produced FDs to some extent. For the future of our case study, we anticipate that readability, intelligibility and evolvability will become crucial. These issues are probably not only to be solved by language and method improvements, but also by tool improvements, most notably smarter visualization and navigation [17]. Hence, our current usage of a general purpose diagramming tool might have to be reconsidered in the rest of the study, to help identify possible improvements at this level too.

Another peculiarity of our settings is the focus on modelling software variability, *vs.* PL variability (see Sec. 2.1). Even though both can be supported by FDs, the former is about representing some objective reality the developers deal with (the variability of the current software assets), whereas the second is about documenting more subjective information (decisions) probably still under debate within product management. Therefore, in the second case, the modelling is more likely to influence the represented information as much as it is influenced by it [15]. The differences in nature between the two variabilities, in the modelling process they imply and in the types of stakeholders they involve, might well justify different modelling constructs and guidelines, as suggested in [19]. The observations reported in this paper, which only concerns the modelling of software variability, should thus not be generalised too hastily to other usages of FDs.

Also specific to our experiment is the OS context. Firstly, OS blurs the boundaries between developers, users and product managers. This might actually attenuate the aforementioned argument that our observations could be biased by the kinds of stakeholders involved in software variability modelling. Maybe this could even extend the relevance of these observations to PL variability modelling. But OS has other characteristics, among which the continuous contribution of software assets by the community of developers. One of the challenges was directly related to this issue (see Sec. 6), hence the question of its generalisation to other contexts. In our opinion, the origin of this challenge resides in the quick and constant evolution of software artifacts, which is by no means specific to OS.

A final validity threat we discuss is the possible bias introduced by the usage of a particular FD language [8]. As we indicated in Sec. 2.2, we used it mainly because of its observed popularity in the research literature and because it appeared to us as representative of current state of the art, considering the richness of its constructs and its formalization. However, doing this, we might have overlooked interesting features of other languages. To alleviate this bias, research efforts could be devoted to model the same problems we did, but with other languages. This will open the way for an empirical comparative evaluation of FD languages, an endeavour that has been quite successful in other application domains of formal methods [2].

9 Conclusion

In this paper, we reported on several challenges encountered during the modelling of the software variability of PloneMeeting, an Open Source software family, with a state-of-the-art feature modelling language. The challenges were illustrated with representative samples of recurrent modelling problems we faced. From the difficulties we experienced and the modelling alternatives we tried, we ended up suggesting language and method improvements as well as more thorough comparative evaluations of modelling notations and techniques. These contributions complement previous work on the formalisation, and more theoretical comparative evaluation, of feature modelling techniques; our ultimate goal being the progress of this field. To avoid overgeneralisation of our conclusions, we eventually discussed the main threats to validity.

Obviously, the story does not end here. The suggested language and method improvements need to be performed and better evaluated; the suggested comparative evaluations as well. The PloneMeeting case study itself must be pursued: further modelling of the software variability is underway, and will be followed by the modelling of product line variability and the evaluation of an approach to automatically cross-check both kinds of variability. Further research might also include comparing feature modelling with other variability modelling paradigms such as decision models.

10 Acknowledgments

We would like to thank Gaëtan Delannay and the other contributors of the PloneGov and PloneMeeting initiatives, as well as our colleagues Pierre-Yves Schobbens, Jean-Christophe Trigaux, Andreas Classen, Hataichanok Unphon and Kim Mens for their valuable comments on this work. This work is sponsored by the Interuniversity Attraction Poles Programme of the Belgian State of Belgian Science Policy under the MoVES project, and partially supported

by the European Commission (FEDER) and Spanish Government under CICYT project Web–Factories (TIN2006-00472).

References

[1] PloneGov. http://www.plonegov.org/.

[2] J.-R. Abrial, E. Börger, and H. Langmaack, editors. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer, 1996.

[3] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Line Conference*, 2005.

[4] D. Benavides, P. Trinidad, and A. Ruiz-Cortez. Automated reasoning on feature models. In LNCS, editor, *17th Conference on Advanced Information Systems Engineering*, pages 491–503, 2005.

[5] C. Cooper. *Building Websites with Plone: An in-depth and comprehensive guide to the Plone content management system*. Packt Publishing, November 2004.

[6] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *IEEE Softw.*, 15(6):37–45, November 1998.

[7] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2), 2005.

[8] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[9] K. Czarnecki and C. H. P. Kim. Cardinality-based feature modeling and constraints: A progress report. In *International Workshop on Software Factories (OOPLSA)*, 2005.

[10] G. Delannay, K. Mens, P. Heymans, P.-Y. Schobbens, and J.-M. Zeippen. PloneGov as an Open Source Product Line. In *Workshop on Open Source Software and Product Lines (OSSPL07), collocated with SPLC*, 2007.

[11] P. Heymans, P.-Y. Schobbens, J.-C. Trigaux, Y. Bontemps, R. Matulevičius, and A. Classen. Evaluating Formal Properties of Feature Diagram Languages. *IET special issue on Language Engineering. To appear*, 2008.

[12] A. Hubaux and P. Heymans. Separating Variability Concerns in a Product Line Re-Engineering Project. In *International workshop on Early Aspects at AOSD*, 2008.

[13] K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.

[14] K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented project line engineering. *IEEE Softw.*, 19(4):58–65, 2002.

[15] J. Krogstie. Using a Semiotic Framework to Evaluate UML for the Development of Models of High Quality. *Unified Modeling Language: System Analysis, Design and Develoment Issues, IDEA Group Publishing*, pages 89–106, 2001.

[16] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In I. C. S. Press, editor, *Proceedings of 15th IEEE International Requirements Engineering Conference*, 2007.

[17] D. Moody. Dealing with "map shock": A systematic approach for managing complexity in requirements modelling. In *12th International Workshop on Requirements Engineering Foundations for Software Quality*, Luxembourg, 2006.

[18] D. E. Perry, S. E. Sim, and S. M. Easterbrook. Case studies for software engineers. In *ICSE*, pages 1045–1046, 2006.

[19] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[20] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148, Minneapolis, Minnesota, USA, September 2006.

[21] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51:456–479, February 2007.

[22] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software – Practice and Experience*, 35(8):705–754, 2005.

[23] F. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[24] J. Van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[25] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, 1997.