A Software Product Line of Feature Modeling Notations and Cross-Tree Constraint Languages

Christoph Seidl[1], Tim Winkelmann[1], Ina Schaefer[1]

Abstract: A Software Product Line (SPL) encompasses a set of closely related software systems in terms of common and variable functionality. On a conceptual level, the entirety of all valid configurations may be captured in a variability model such as a feature model with additional cross-tree constraints. Even though variability models are essential for specifying configuration knowledge, various notations for feature models and cross-tree constraints exist, which increases implementation effort when having to realize new tools for a different language. In this paper, we provide remedy to this problem by introducing an SPL to generate different variants of feature modeling notations and cross-tree constraint languages. We base our approach on the state of the art in various works and surveys on feature modeling to create a family of feature modeling notations with similar expressiveness as the original approaches. For our findings, we provide both conceptual configuration knowledge as well as a generative model-based realization. We further demonstrate the feasibility of our approach by generating feature modeling notations similar to those of various publications.

1 Introduction

A Software Product Line (SPL) encompasses a set of closely related software systems in terms of common and variable functionality. On a conceptual level, configuration knowledge may be captured in a *variability model* to describe configuration rules for all valid products. The most commonly used type of variability models are *feature models* [Ka90], which arrange *features* as configurable units along a tree-structured decomposition hierarchy. Commonly, language constructs are used to represent *optional* and *mandatory* features as well as *alternative-groups* allowing selection of exactly one feature and *or-groups* permitting selection of one or more features. Furthermore, cross-tree constraints [Ba05] may be used to further restrain configuration options.

Ever since the original introduction of feature models [Ka90], a great number of extensions has been made to the original notation to address various needs, such as attributes for features with finite and infinite domains [Cz02, CHE05, KOD10], cardinalities for groups and individual features [Ri02, CHE05, ME08, SSA14a] or configurable feature versions [SSA14a, SSA14c]. Likewise, languages for cross-tree constraints may be represented by different means using various subsets of propositional logic [Ba05] or the OCL [CE00, Cz02] (Object Constraint Language) as well as textual formulations [Ba05, KOD10] and graphical representation as additional edges in the feature model [HSVM00, SLW12]. Through these extensions for feature models and cross-tree constraints, a wide variety of different concerns of configuration problems can be addressed.

[1] Software Engineering Institute, Technische Universität Braunschweig, E-Mail: {c.seidl,t.winkelmann,i.schaefer}@tu-bs.de

However, the cluttering of different languages also entails problems in practice as, e.g., implementation effort is increased when having to realize new tools for a different language, which results in hampered progress and, presumably, less robust software.

Despite the differences in the various languages for feature models and cross-tree constraints, the languages encompass a significant level of commonality. In this paper, we exploit this fact to make a first step towards solving the aforementioned problems by devising an SPL to generate feature modeling notations and cross-tree constraint languages with a selected set of language constructs. We bootstrap feature modeling to conceptually capture the configuration knowledge of the software family. We further use SPL techniques to allow configuration of various different concrete languages for feature models and cross-tree constraints. We supply both theoretical background as to the capabilities of the different language constructs as well as technical realizations of the different concrete languages through generative model-based development.

With these contributions, it is possible to derive variants of feature modeling notations and cross-tree constraint languages that are similar to those of existing approaches and to treat them with procedures common in SPL engineering (e.g., family-based analyses). Furthermore, new combinations of feature modeling language constructs may yield previously non-existing notations to address the individual characteristics of specific use cases. In the future, this approach may be used to support data exchange between different notations by transforming one notation of a source system to an (at least) equally expressive notation with different characteristics of a target system.

The rest of this paper is structured as follows: Section 2 provides the criteria we used to select feature modeling notations we consider for our SPL. Section 3 analyzes 23 individual approaches from the state of the art in feature modeling (including constraint languages) as well as 4 surveys on feature modeling notations and categorizes the respective approaches by a number of distinctive characteristics. Section 4 presents our family of feature models and constraint languages that subsumes all of the analyzed approaches in expressiveness and even allows creation of previously non-existent notations. Section 5 demonstrates feasibility of our work by first applying the implementation of the presented feature modeling family to generate multiple variants with different expressiveness and then recreating the available examples of the inspected approaches. Finally, Section 6 closes with a conclusion and an outlook to future work.

2 Considered Work

A large number of similar yet different feature modeling notations exists which address a wide variety of different concerns. Developers have to find the right tool with the right notation for their respective SPL project. To help developers in making a conscious decision, an overview of existing notations and their dependencies is needed. As the different characteristics of feature modeling notations can themselves be represented as features and the dependencies can be expressed using constraints, we create an SPL that covers the variability of the inspected modeling notations. For this purpose, we define a family of feature modeling notations including cross-tree constraint languages to express dependencies between the elements of a feature model. However, we do not claim completeness with regard to expressiveness or other properties such as succinctness or naturalness [SHT06] for *all* feature modeling notations. Nevertheless, we do cover a wide range of approaches used in practice [Ka90, Ka98, GFA98, HSVM00, vGBS01, Ri02, EBB05, CE00, Cz02, CHE04, BTRC05] as well as specific special-purpose extensions [ME08, SLW12, KSS13, SSA14a].

To select suitable feature modeling notations to analyze as basis for the presented software family, we applied the following criteria for selection: First, we included those feature modeling notations that have a particular high impact on further development. For this purpose, we considered all approaches included in Kang's keynote presentation from VaMoS'10 on 20 years of feature modeling [Ka10] as they are fundamental for many further feature modeling notations [Ka90, Ka98, GFA98, HSVM00, vGBS01, Ri02, EBB05, CE00, Cz02, CHE04, BTRC05]. We further included surveys that analyze state of the art in feature modeling [Jé12, BSRC10, SHT06, CHE04] to determine relevant feature modeling approaches. Second, we included approaches that are representative for various special-purpose extensions [ME08, SLW12, KSS13, SSA14a] in feature modeling. Third, we included textual variability languages, such as TVL [CBH11], Familiar [Ac13] and Clafer [BCW11], which can be represented with a meta-model similar to feature models. For work from the last 10 years, we mainly considered those extensions that explicitly provide a meta-model or grammar for their language and that introduce new concepts to feature models.

Existing surveys on feature modeling notations have presented contributions that are closely related to our work: Schobbens et al. [SHT06] also analyzed a great number of the high impact feature modeling notations in a formal way. During their analyses, they devised a language called *Varied Feature Diagrams (VFD)* [Sc07]. For VFD, they used expressive and succinct elements of feature modeling notations to build a combined modeling notation. That notation is as expressive and succinct as the other analyzed notations. Their approach is different from ours as we generate variants of feature modeling notations with capabilities tailored to the respective intended use instead of creating one monolithic notation for all use cases. Dhungana et al. [Dh13] allow the use of various different variability modeling notations and transform them into a uniform notion of configuration options. Hence, their approach is practical for the configuration process of different employed variability modeling notations where ours aims towards the modeling process. Hubaux et al. [HTH13] analyzed feature diagram languages with regard to separation and composition of concerns. They identified important concerns and purposes of feature diagram languages, how they are separated and composed in existing SPL approaches and other characteristics. Different notations we discuss in this paper have a direct impact on separation and composition of SPLs. Jézéquel [Jé12] surveys several modeling notations and explores the combination of these notations with artifacts in the product generation process. Lichter et al. [Li03] compare a number of feature modeling notations with a focus on the process and required input to make a particular feature modeling notation useful. While they acknowledge large commonalities but also differences of notations, they do not build an SPL to create different variants of feature modeling notations. Eichelberger and Schmid [ES13, ES14] present an analysis of textual variability languages. They provide an overview of the commonalities and differences of the notations in order to provide information on the evolution of textual variability modeling languages and identifying common weaknesses for future research.

We improve over the mentioned literature reviews [Jé12, Li03, BSRC10, Ka10, Sc07] by not just listing and discussing different characteristics of feature modeling notations but by further presenting an SPL and a model-based realization to generate individual variants. The idea of our work is that the provided SPL can be used to create variants for representing configuration knowledge in a notation tailored to the concrete use case. However, at present, proofs for semantic equivalence of arbitrary variants are outside the scope of the paper. In the future, appropriate tools should be generated, such as an editor that supports the respective notations. As basis for this SPL, we selected 23 different feature modeling notations we analyze for distinct characteristics in the next section.

3 Analysis

Using the criteria presented in Section 2, we determined 23 different approaches for feature modeling notations and constraint languages to analyze regarding their common and distinctive characteristics as basis for a software family. Table 1 summarizes our findings and the following sections elaborate on characteristics of the analyzed approaches regarding notational concepts of the employed feature modeling and constraint languages.

3.1 Feature Modeling Notations

In the upper part of Table 1, we provide information on different language constructs provided by the examined approaches, which are explained in the following.

Mandatory Features represent commonalities that have to be included in a configuration if their parent feature is selected. All examined approaches support this language construct.

Optional Features represent variabilities that may or may not be included in a configuration. All examined approaches support this language construct.

Feature Cardinality specifies a minimum and maximum number for how often a feature may be selected as [m..n]. When using 1 as maximum cardinality [Cz02, CHE04, Ri02, ME08, SLW12, SSA14a], feature cardinality may be perceived as alternative to the explicit variation type for mandatory features (i.e., [1..1]) and optional features (i.e., [0..1]). Furthermore, Czarnecki et al. [Cz02, CHE04] use feature cardinalities to be able to represent multiple instances (e.g., [2..5]) of one and the same feature as *cloned features* by allowing maximum cardinalities greater than 1.

Attributes are named variables of features [Ri02, Cz02, CHE04, BTRC05] that refine configuration options so that, besides selection of features, concrete values for attributes may be chosen. Czarnecki et al. [Cz02, CHE04] and Benavides et al. [BTRC05] assign a specific *type* to the attributes, which specifies permissible values. In the literature, types of attributes are typically categorized into *discrete* (*finite* or *infinite*) and *continuous* domains.

Feature Versions include variability in time in feature models [SSA14a, ME08]. Mitschke et al. [ME08] support two versions per feature representing the state of the feature model's structure and its associated implementation but do not allow using them as configurable units. Seidl et al. [SSA14a, SSA14c] allow specification of multiple feature versions with interdependencies to make feature versions a configurable unit. Configurable versions may not adequately be represented using attributes as their relation cannot be specified properly.

Layers of feature models provide a separation of concerns for different sources of variability. Kang et al. [Ka98] use layers for *Capability*, *Operating Environment*, *Domain Technology* and *Implementation Technique*. Each layer may contain a set of separate feature models with relations to feature models of other layers. This increases the reuse of feature models and supports scalability.

External Features allow referencing of features that are defined in other feature models [vGBS01]. For example, this may be used in combination with layers of feature models when referencing features of other feature models [BCW11, Ab10, CBH11, Ro11, Ac13].

Binding Times specify at which time a feature may be or has to be configured. Typical binding times are at *compile time* or *run time* [GFA98, vGBS01, B3]. Griss et al. [GFA98] use attributes in the features to describe the binding time. Van Gurp et al. [vGBS01] use a label on the connector between features to distinguish the binding time.

Resource Mapping allows association of various resources with the features in a feature model [SLW12, KSS13, Th11]. Schroeter et al. [SLW12] provide a mapping of features to views, which show only selective parts of a feature model for different stakeholders of the feature model. Kowal et al. [KSS13] map priorities for the configuration and specific hardware to the features.

Alternative-Groups allow selection of exactly one of the contained features, which makes them mutually exclusive. All examined approaches support this language construct.

Or-Groups allow selection of at least one of the contained features. With the exception of Kang et al. [Ka90], all examined approaches support this language construct.

Group Cardinality specifies the minimum and maximum number of selectable features in that group as [m..n]. Hence, it may be perceived as an alternative to the explicit variation type of groups as alternative-groups (i.e., [1..1]) and or-groups (i.e., [0..n] for groups with *n* members) [Ri02, CHE04, SLW12, SSA14a]. In contrast to the explicit variation types, group cardinality supports further restrictions on selections in a group (e.g., [2..5]).

Multiple Groups describe the possibility that a feature can have more than one child group, e.g., a feature that has two alternative-groups. Many notations do not explicitly state whether they support multiple groups or not. Czarnecki and Eisenecker [CE00] appear to be the first who explicitly support multiple groups.

		Kang et al. 1990 [KCH+90]	Kang et al. 1998 [KKL+98]	Griss et al. 1998 [GFA98]	Hein et al. 2000 [HSVM00]	Czarnecki et al. 2000 [CE00]	Van Gurp et al. 2001 [vGBS01]	Riebisch et al. 2002 [RBSP02]	Czarnecki et al. 2002 [CBUE02]	Czarnecki et al. 2004 [CHE04]	Batory 2005 [Bat05]	Eriksson et al. 2005 [EBB05]	Benavides et al. 2005 [BTRC05]	Schobbens et al. 2007 [SHTB07]	Mitschke et al. 2008 [ME08]	Bąk et al. 2011 [BCW11]	Abele et al. 2010 [APS+10]	Classen et al. 2011 [CBH11]	Thüm et al. 2011 [TKES11]	Rosenmüller et al. 2011 [RSTS11]	Schroeter et al. 2012 [SLW12]	Kowal et al. 2013 [KSS13]	Acher et al. 2013 [ACLF13]	Seidl et al. 2014 [SSA14a]
Feature Modeling Notation	**Mandatory Features**	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	Optional Features	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	Feature Cardinality	-	-	-	-	-	-	o	+	+	-	-	-	+	o	+	+	+	-	-	o	-	-	o
	Attributes	-	-	+	-	-	-	o	+	+	-	-	+	-	-	+	+	+	-	-	-	-	-	-
	Feature Versions	-	-	-	-	-	-	-	-	-	-	-	-	-	o	-	-	-	-	-	-	-	-	+
	Layers	-	+	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	-	-	+	-
	External Features	-	-	-	-	-	+	-	-	-	-	-	-	-	-	+	+	+	-	+	-	-	+	(+)
	Binding Times	-	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	-
	Resource Mapping	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	-	+	+	-	-
	Alternative-Groups	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	Or-Groups	-	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	Group Cardinality	-	-	-	-	-	-	+	-	+	-	-	-	+	+	+	+	+	-	-	+	-	-	+
	Multiple Groups	(-)	(-)	(-)	(-)	+	+	+	+	+	-	(+)	+	(-)	(-)	+	+	(-)	-	+	(-)	(-)	(-)	+
Constraint Language	**Expressiveness**	R	R	R	R	O	R	O	O	-	P	R	-	P	R	P+	P	P+	P	P	R	P	P	P+
	Representation	T	T	G	G	T	G	G,T	T	-	T	G	-	T	T	T	G,T	T	T	T	G	T	T	T

R: Requires/Excludes, O: OCL, P: Propositional Logic, T: Textual, G: Graphical

Table 1: Distinctive characteristics of the inspected feature modeling notations.

3.2 Constraint Languages

The inspected feature modeling approaches utilize various constraint languages. They differ in their expressiveness and their representation as presented in the bottom part of Table 1.

Expressiveness of constraint languages is determined by the employed formalism and its utilized language constructs. For one, mere *requires* and *excludes* relations (R) may be specified [Ka90, GFA98, HSVM00, vGBS01, EBB05, SLW12]. Furthermore, the OCL (O) is used in some approaches [CE00, Cz02]. In addition, it is possible to utilize propositional logic (P). Depending on the concrete work, different subsets of Boolean operators are utilized to specify constraints over features [KSS13, SSA14a]. In addition, new language constructs are introduced for special purposes (P+), e.g., to compare attribute values [KOD10] or to express constraints over feature versions [SSA14a].

Representation of constraints is either graphical or textual. With a graphical (G) representation, additional edges are added between two features to express requires or excludes relationships [GFA98, HSVM00, vGBS01, EBB05, SLW12]. Textual representations (T) may be employed for a wider range of formalism, such as requires and excludes relationships [Ka90], OCL [CE00, Cz02] or subsets of propositional logic [KSS13, SSA14a]. In the latter case, there is a further distinction on how Boolean operators are represented as they may use logical symbols (e.g., $\wedge$, $\vee$), a verbal representation (e.g., *and*, *or*) or a representation known from various programming languages such as Java or C++ (e.g., `&&`, `||`). In some cases [Ri02, Ab10], both a textual and a graphical representation of constraints is provided.

4 Feature Modeling Family

From the results of the analysis in Section 3, we define a software family of feature modeling notations and constraint languages, which subsumes the individual approaches examined in Section 3. Furthermore, it is possible to generate variants with combinations of language constructs that currently do not exist in the literature or in practice.

We use a feature model to describe all valid configurations of the family in terms of configuration knowledge. Due to its size, the graphical representation of the feature model is split up over multiple figures. Figure 1 shows an overview of the top-level features of the family with `FeatureModel` describing the configuration options of the feature modeling notation and `ConstraintLanguage` capturing the configuration options for the cross-tree constraint language. Both these features are refined and described in detail in the following sections.

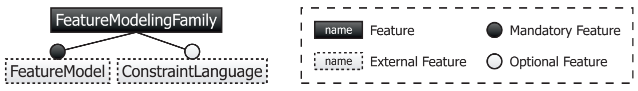

Figure 1: Top-level features of the feature modeling family. `FeatureModel` and `ConstraintLanguage` are defined in Figure 2 and Figure 3, respectively.

4.1 Variability of Feature Modeling Notations

Figure 2 shows a refined view of the feature model branch describing configuration options for the various feature modeling notations. We modeled both `Features` and `Groups` to be mandatory parts of each feature modeling notation. Features represent their type by either using a `FeatureCardinality` (e.g., [1..1] for mandatory) or an explicit `FeatureVariationType` (i.e., `OptionalFeatures` or `MandatoryFeatures`). When using feature cardinalities, it is further possible to allow `ClonedFeatures` if a feature can be instantiated multiple times. In addition, it is possible to explicitly allow `UnlimitedFeatures` by providing an unbounded maximum cardinality (using *) instead of an integer value. As the latter case implicitly depends on cloned features being enabled, we introduced constraint (1). Furthermore, it is possible to enable `External` to reference features defined in a different feature model, e.g., to realize feature layers [Ka98]. Using `Resources` allows association of features with arbitrary resources. Finally, enabling `BindingTimes` allows assigning a binding time to a feature, e.g., *compile time*.

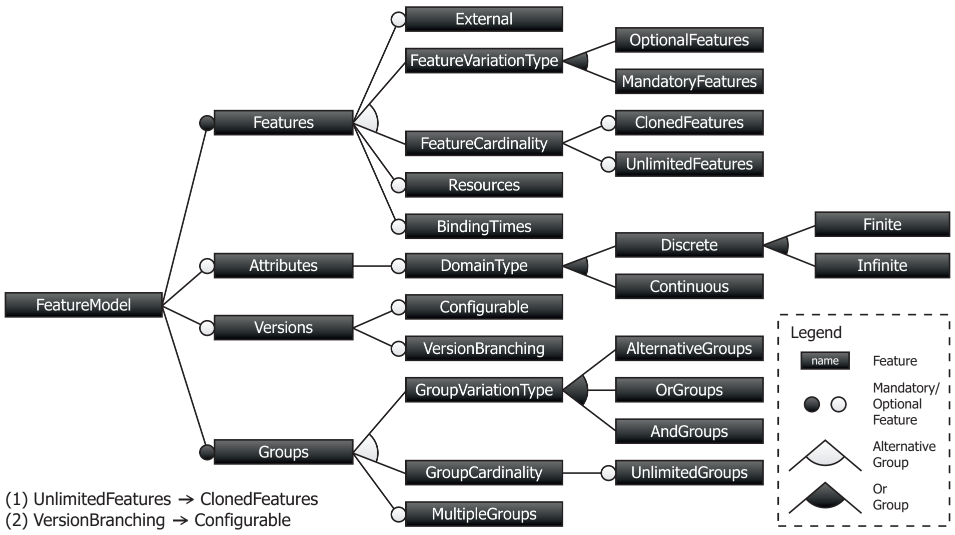

Figure 2: Feature model branch describing configuration options for feature modeling notations.

Furthermore, `Attributes` may be included in the feature model notation. Optionally, attributes have a `DomainType`, which specifies the domain for an attribute as either `Discrete` (e.g., enumerations, integer numbers or strings) or `Continuous` (e.g., floating point numbers). A discrete domain type may further be either `Finite` (e.g., enumerations) or `Infinite` (e.g., integer numbers or strings).

Additionally, it is possible to enable feature `Versions` to support variability in time. It can be decided whether versions can be used as `Configurable` units (as in [SSA14a]) or not (as in [ME08]). Versions are arranged along a chronological development line that is assumed to be linear unless `VersionBranching` is selected, which allows different branches, which depends on configurable versions so that we introduced constraint (2).

Similarly to features, groups represent their type either as `GroupCardinality` or as explicit `GroupVariationType`. To subsume the expressiveness of the inspected approaches, it would have been possible to model `AlternativeGroups` as a mandatory feature and `OrGroups` as an optional feature. We decided to use an or-group instead to give more liberty in variant derivation and also allow feature models that do not use alternative-groups. When using group cardinalities, it is possible to enable `UnlimitedGroups`, which permit an arbitrary number of features to be selected by providing an unbounded maximum cardinality (using *) instead of an integer value. Generally, only a single group is allowed as child of a feature unless `MultipleGroups` is selected.

4.2 Variability of Constraint Languages

Figure 3 shows a refined view of the feature model branch describing configuration options of the various cross-tree constraint languages. We designed the feature model of constraint languages to support `Propositional` logic or `OCL` as well as textual and graphical representations.

For propositional logic, various options for different language constructs exist (grouped by their respective number of operands). A number of `AtomicConstructs` is provided: The mandatory `FeaturePresence` checks whether a specified feature is part of a configuration. `AttributeRestrictions` allow comparison of attribute values using various arithmetic operators (`LT` ($<$), `LEQ` ($\leq$), `GEQ` ($\geq$) and `GT` ($>$)), string operators (`SUBS` as substring comparison) or those used in both contexts (`EQ` ($=$) and `NEQ` ($\neq$)).

Optionally, `VersionRestrictions` [SSA14a] may be selected with `VersionRangeRestrictions` allowing dependence on an interval of versions, `RelativeVersionRestrictions` specifying a valid set of versions in relation to a given version and `ConditionalVersionRestrictions` allowing evaluation of the former constructs for configurable versions only if their respective containing feature is present in a configuration. As restrictions for attributes and versions may only be specified if the respective elements are part of the notation, constraints (3) and (4) were added.

The sole child of `UnaryConstructs` is the feature `Not` ($\neg A$) as logical negation, which may be deselected by not selecting its parent. Furthermore, various `BinaryConstructs` are provided as known from Boolean algebra:

- `Or`: $A \vee B$ (logical or)
- `And`: $A \wedge B$ (logical and)
- `Xor`: $(A \oplus B) \equiv ((A \wedge \neg B) \vee (\neg A \wedge B))$
- `Implies`: $(A \rightarrow B) \equiv (\neg A \vee B)$[1]
- `Equivalent`: $(A \equiv B) \equiv ((A \rightarrow B) \wedge (B \rightarrow A))$
- `Excludes`: $(A \texttt{ excludes } B) \equiv \neg(A \wedge B)$

[1] We did not define a `Requires` feature as it is semantically equivalent to the already existing `Implies`.

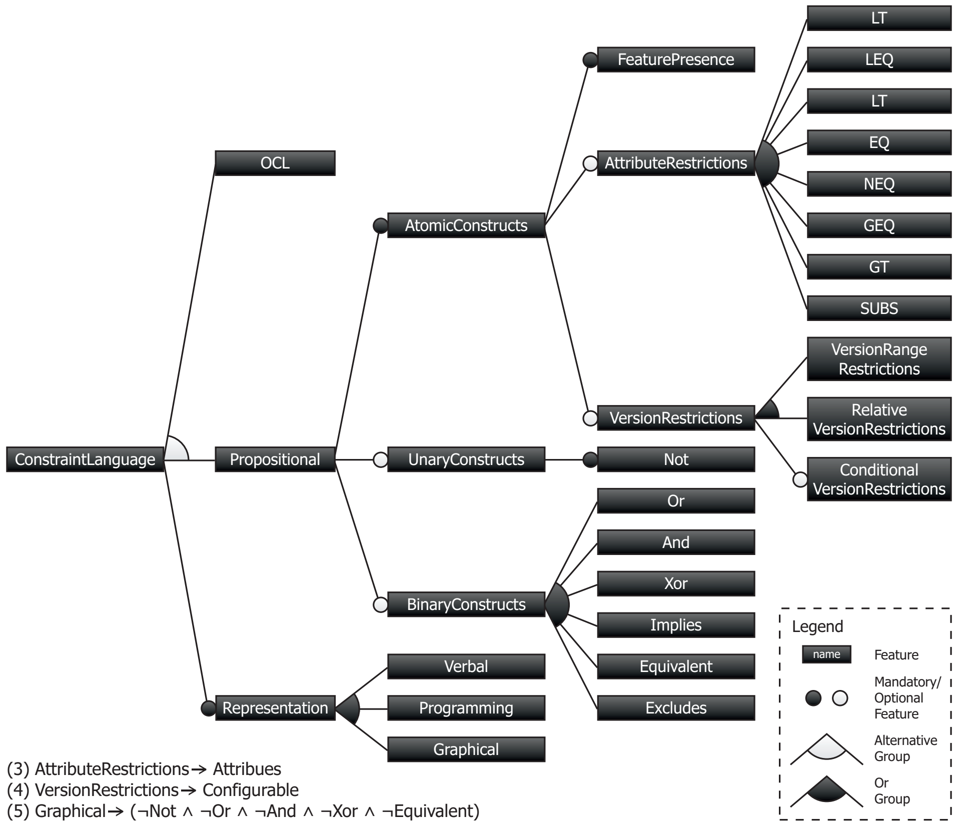

Figure 3: Feature model branch describing configuration options for cross-tree constraint languages.

The expressiveness of the constraint language may differ with the selection of supported constructs: For example, both selections {`Or`, `Not`} and {`Implies`, `Excludes`} result in a language that is complete with regard to Boolean logic. In contrast, when only selecting `Not` as construct, the expressiveness of the resulting cross-tree constraint language is severely limited.

It is possible to choose different representations for these constructs (`Representation`) as either textual or graphical: With a `Verbal` representation, textual literal names are used for constructs (e.g., *and*, *or*). With a `Programming` representation, textual operands similar to those used in Java or C++ are used (e.g., `&&`, `||`)[2]. Alternatively, it is also possible to choose a `Graphical` representation where constraints are added as additional edges between features to the visual representation of the feature model. This type of representation is only capable of visualizing implications and exclusions so that constraint (5) was added to exclude all other constructs when the graphical representation is selected. The type

[2] With the implementation of the feature modeling family in mind, we did not include a constraint representation that uses logical operators from Boolean logic (e.g., $\wedge$, $\vee$) as those symbols cannot be typed on a keyboard.

of representation for constraints is further relevant for a generation of variants because a textual representation requires a grammar for the language to be supplied (see Section 5).

5 Case Study

To demonstrate the feasibility of our approach, we performed a case study using the presented family of feature modeling notations and constraint languages.[3] For this purpose, we provide a prototypical model-based realization of the aforementioned family of feature models and constraint languages in the form of an SPL as depicted in Figure 2 and Figure 3. This SPL may be used to generate variants of the underlying meta-models and grammars for individual feature model notations and cross-tree constraint languages regarding the described configuration options. Within the case study, we are particularly interested in answering two research questions:

RQ1 Is it possible to derive meta-models for feature models and constraint languages that are as expressive as the approaches analyzed in Section 3?

RQ2 Does the family of feature modeling notations support derivation of notations that have not been devised before?

We employ the transformational variability realization mechanism delta modeling [Sc10] to generate different variants of the family. In delta modeling, a *base variant* of a system is transformed to a *target variant* by applying a number of *delta modules* that each specify a set of coherent transformations defined as sequence of *delta operations*. A *delta language* provides the delta operations available to alter a *source language* by adding, modifying and removing elements. A variant is derived by selecting a valid subset of delta modules (e.g., using a feature model with a mapping to delta modules) and applying the delta modules in a suitable order to generate the intended target variant by transforming the base variant.

As base variant for the parts of the software family regarding the feature model and the constraint language based on propositional logic, we use the meta-models for Hyper Feature Models and their version-aware constraint language as used in our previous work [SSA14a, SSA14c], which provide language constructs as described by the last column of Table 1. Figure 4 depicts a representative excerpt of the base meta-model for feature models defined using Ecore[4] of the Eclipse Modeling Framework (EMF). Similarly, the base meta-model for constraint languages in propositional logic is also defined in EMF Ecore but further uses a concrete syntax to define a textual language using the tool EMFText[5]. For OCL, we use the meta-model and textual representation provided by the DresdenOCL toolkit[6], which is again based on EMF Ecore.

To make these artifacts subject to variability in delta modeling, we defined delta languages for both Ecore meta-models and concrete syntax files of EMFText using the delta language

[3] `https://fusionforge.zih.tu-dresden.de/projects/snowflake`
[4] `http://eclipse.org/emf`
[5] `http://emftext.org`
[6] `http://dresden-ocl.org`

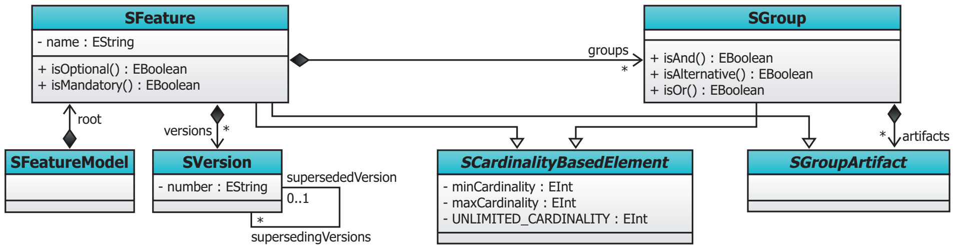

Figure 4: Excerpt from the base variant for the Ecore meta-model of the feature model family.

generation framework DeltaEcore[7] [SSA14b]. We further defined 58 delta modules that realize changes to accommodate for the different language constructs of the feature model and constraint language families when generating variants. We assigned delta modules to (combinations of) features of Figure 2 and Figure 3 so that variants can be generated by selecting a valid configuration of features, determining the respective delta modules and applying them in an automatically determined suitable order.

We defined configurations to represent the distinctive characteristics of each analyzed work by selecting features from our family of feature modeling notations that reflect the entries in each column of Table 1. We used these configurations to generate a variant for each inspected work consisting of the meta-models for the feature model and its constraint language as well as the concrete syntax file of the textual constraint language (if applicable). We inspected the generated variants for conformance with the selected configurations as well as their expressiveness with regard to the included distinctive characteristics. We used the generated variants of the meta-model and the constraint language to recreate the examples presented in each of the analyzed works[8]. Figure 5 shows an example of a variant of the meta-model for feature models for the configuration that consists of the features `FeatureModelingFamily`, `FeatureModel`, `Features`, `External`, `FeatureVariationType`, `OptionalFeatures`, `MandatoryFeatures`, `Groups`, `Group-VariationType`, `AlternativeGroups`, `OrGroups`, `AndGroups` and `MultipleGroups`. This variant resembles the notation used for the diagrams presented in Figure 1, Figure 2 and Figure 3.

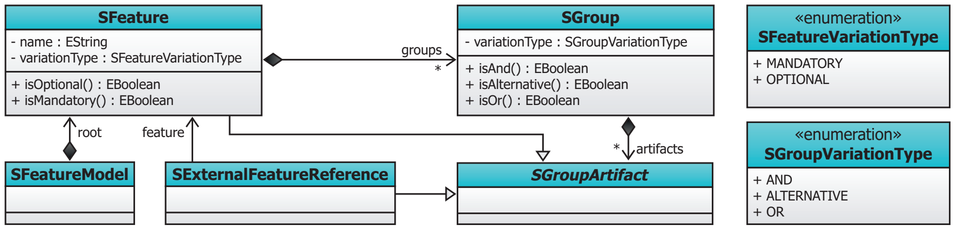

Figure 5: Excerpt from an example variant for the Ecore meta-model of the feature model family resembling the feature model notation used for the diagrams of this paper.

[7] `http://deltaecore.org`

[8] For papers, we recreated all presented examples. However, [CE00] is a book with over 800 pages so that we focused on creating a sample of all presented feature models.

As a result of generating specifically tailored variants of feature model notations and cross-tree constraint languages, the majority of concepts could be expressed directly by dedicated language constructs. In addition, we used external features linking different feature models to realize layers [Ka98]. However, we could not directly represent calculated attribute values [BTRC05] but had to substitute constraints on the attributes demanding respective values as a workaround. Finally, we had to create mock up models for the externally defined resources (e.g., hardware, views) to realize resource mappings [SLW12, KSS13], which are otherwise supplied along with a particular SPL. Using these techniques, we were able to capture all information presented in the original work by employing a variant generated from the feature modeling family. As a conclusion, we were able to answer *RQ1* positively as we could generate feature model notations and cross-tree constraint languages with similar expressiveness as the inspected approaches with regard to the information available in the respective publications.

In addition, we defined configurations for feature modeling notations that have a combination of language constructs that, to our knowledge, did not exist, yet. For example, we derived the variant for the feature models presented as diagrams in this paper, which includes multiple groups and external features (as presented in Figure 5). Furthermore, we generated previously non-existent variants, such as a variant with cardinality-based features as well as binding times and resource mappings for features. As a result, we were able to answer *RQ2* positively.

6 Conclusion

In this paper, we presented a family of feature modeling notations and constraint languages that encompasses various similar, yet different notations in order to generate specifically tailored variants of feature model and cross-tree constraint notations. As basis, we analyzed state of the art in feature modeling notations and classified 23 approaches by the notational concepts they offer. From these findings, we assembled conceptual configuration knowledge within feature models for a family of feature modeling notations and cross-tree constraint languages. We provided a model-based realization of this family and used it in a case study to demonstrate feasibility of our approach by generating variants with expressiveness similar to the analyzed approaches as well as previously non-existent notations. Using our work, it is possible to bootstrap SPL techniques to use them on feature models and their constraint languages to generate variants of feature model and cross-tree constraint notations according to a particular selection of language constructs. This is beneficial when depending on a notation with specific capabilities and may further be useful when transforming configuration knowledge specified in different feature modeling notations, analyzing configuration options of various different feature models or integrating configuration knowledge from various sources with different notations.

In our future work, we will extend the provided SPL with variability of analyses techniques, possible graphical representations, the generation of adequate tools for the variants as well as support for different SPL implementation techniques. A configuration of this feature modeling notation may determine which solvers (e.g., SAT, BDD, CSP) can be used, which analysis techniques are available and also which other dependencies need to be considered for an implementation (e.g., if a chosen feature modeling notation restricts the choice for an SPL implementation techniques). Additionally, we will analyze effects of transforming models conforming to one variant of the feature model family to conform to another variant by substituting language constructs. As far as feasible, we will provide an implementation based on model transformation to allow data exchange between different variants of the family for feature modeling notations.

Acknowledgments

This work was partially supported by the DFG (German Research Foundation) under grants SCHA1635/2-1 and SCHA1635/4-1 and by the European Commission within the project HyVar (grant agreement H2020-644298).

References

[Ab10] Abele, Andreas; Papadopoulos, Yiannis; Servat, David; Törngren, Martin; Weber, Matthias: The CVM Framework-A Prototype Tool for Compositional Variability Management. VaMoS, 10:101–105, 2010.

[Ac13] Acher, Mathieu; Collet, Philippe; Lahire, Philippe; France, Robert B.: FAMILIAR: A Domain-specific Language for Large Scale Management of Feature Models. Sci. Comput. Program., 78(6):657–681, June 2013.

[B3] Bürdek, Johannes; Lity, Sascha; Lochau, Malte; Berens, Markus; Goltz, Ursula; Schürr, Andy: Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints. In: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS '14, 2013.

[Ba05] Batory, D.: Feature Models, Grammars, and Propositional Formulas. Software Product Lines, 2005.

[BCW11] Bak, Kacper; Czarnecki, Krzysztof; Wasowski, Andrzej: Feature and Meta-models in Clafer: Mixed, Specialized, and Coupled. In: Proceedings of the Third International Conference on Software Language Engineering. SLE'10, Springer-Verlag, Berlin, Heidelberg, pp. 102–122, 2011.

[BSRC10] Benavides, David; Segura, Sergio; Ruiz-Cortés, Antonio: Automated Analysis of Feature Models 20 Years Later: A Literature Review. Information Systems, 2010.

[BTRC05] Benavides, David; Trinidad, Pablo; Ruiz-Cortés, Antonio: Automated Reasoning on Feature Models. In: Proceedings of the 17th International Conference on Advanced Information Systems Engineering. CAiSE'05. Springer, 2005.

[CBH11] Classen, Andreas; Boucher, Quentin; Heymans, Patrick: A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL. Science of Computer Programming, 2011.

[CE00] Czarnecki, Krzysztof; Eisenecker, Ulrich W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CHE04] Czarnecki, Krzysztof; Helsen, Simon; Eisenecker, Ulrich: Staged Configuration Using Feature Models. In (Nord, RobertL., ed.): Software Product Lines, volume 3154 of Lecture Notes in Computer Science, pp. 266–283. Springer Berlin Heidelberg, 2004.

[CHE05] Czarnecki, Krzysztof; Helsen, Simon; Eisenecker, Ulrich: Formalizing Cardinality-Based Feature Models and their Specialization. In: Software Process: Improvement and Practice. 2005.

[Cz02] Czarnecki, Krzysztof; Bednasch, Thomas; Unger, Peter; Eisenecker, Ulrich: Generative Programming for Embedded Software: An Industrial Experience Report. In (Batory, Don; Consel, Charles; Taha, Walid, eds): Generative Programming and Component Engineering, Lecture Notes in Computer Science. Springer, 2002.

[Dh13] Dhungana, Deepak; Seichter, Dominik; Botterweck, Goetz; Rabiser, Rick; Grünbacher, Paul; Benavides, David; Galindo, José A.: Integrating Heterogeneous Variability Modeling Approaches with Invar. In: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems. VaMoS '13, 2013.

[EBB05] Eriksson, Magnus; Börstler, Jürgen; Borg, Kjell: The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In (Obbink, Henk; Pohl, Klaus, eds): Software Product Lines, volume 3714 of Lecture Notes in Computer Science, pp. 33–44. Springer Berlin Heidelberg, 2005.

[ES13] Eichelberger, Holger; Schmid, Klaus: A Systematic Analysis of Textual Variability Modeling Languages. In: Proceedings of the 17th International Software Product Line Conference. SPLC '13, ACM, New York, NY, USA, pp. 12–21, 2013.

[ES14] Eichelberger, Holger; Schmid, Klaus: Mapping the design-space of textual variability modeling languages: a refined analysis. International Journal on Software Tools for Technology Transfer, pp. 1–26, 2014.

[GFA98] Griss, M. L.; Favaro, J.; Alessandro, M. d': Integrating Feature Modeling with the RSEB. In: Proceedings of the 5th International Conference on Software Reuse. ICSR '98, IEEE Computer Society, Washington, DC, USA, 1998.

[HSVM00] Hein, Andreas; Schlick, Michael; Vinga-Martins, Renato: Applying Feature Models in Industrial Settings. In: Proceedings of the First Conference on Software Product Lines: Experience and Research Directions. Kluwer Academic Publishers, 2000.

[HTH13] Hubaux, Arnaud; Tun, Thein Than; Heymans, Patrick: Separation of Concerns in Feature Diagram Languages: A Systematic Survey. ACM Comput. Surv., 45(4):51:1–51:23, August 2013.

[Jé12] Jézéquel, Jean-Marc: Model-Driven Engineering for Software Product Lines. ISRN Software Engineering, 2012.

[Ka90] Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A. S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University SEI, 1990.

[Ka98] Kang, Kyo C.; Kim, Sajoong; Lee, Jaejoon; Kim, Kijoo; Shin, Euiseob; Huh, Moonhang: FORM: A Feature-Oriented Reuse Method with Domain-specific Reference Architectures. Ann. Softw. Eng., 5:143–168, January 1998.

[Ka10] Kang, Kyo: , FODA: Twenty Years of Perspective on Feature Modeling. Keynote at the 4th International Workshop on Variability Modelling of Software-Intensive Systems, 2010.

[KOD10] Karataş, Ahmet Serkan; Oğuztüzün, Halit; Doğru, Ali: Global Constraints on Feature Models. In: Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming, CP'10, pp. 537–551. Springer-Verlag, Berlin, Heidelberg, 2010.

[KSS13] Kowal, Matthias; Schulze, Sandro; Schaefer, Ina: Towards Efficient SPL Testing by Variant Reduction. In: Proceedings of the 4th International Workshop on Variability & Composition. VariComp '13, ACM, New York, NY, USA, pp. 1–6, 2013.

[Li03] Lichter, Horst; von der Maßen, Thomas; Nyßen, Alexander; Weiler, Thomas: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung. Technical Report AIB-2003-07, Department of Computer Science, RWTH Aachen, July 2003.

[ME08] Mitschke, R.; Eichberg, M.: Supporting the Evolution of Software Product Lines. In: ECMDA Traceability Workshop. ECMA-TW, 2008.

[Ri02] Riebisch, M.; Böllert, K.; Streitferdt, D.; Philippow, I.: Extending Feature Diagrams with UML Multiplicities. In: 6th World Conference on Integrated Design & Process Technology (IDPT2002). June 2002.

[Ro11] Rosenmüller, Marko; Siegmund, Norbert; Thüm, Thomas; Saake, Gunter: Multi-dimensional Variability Modeling. In: Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems. VaMoS '11, ACM, New York, NY, USA, pp. 11–20, 2011.

[Sc07] Schobbens, Pierre-Yves; Heymans, Patrick; Trigaux, Jean-Christophe; Bontemps, Yves: Generic Semantics of Feature Diagrams. Comput. Netw., 51(2):456–479, February 2007.

[Sc10] Schaefer, Ina; Bettini, Lorenzo; Bono, Viviana; Damiani, Ferruccio; Tanzarella, Nico: Delta-Oriented Programming of Software Product Lines. In: Software Product Lines: Going Beyond, pp. 77–91. Springer, 2010.

[SHT06] Schobbens, Pierre-Yves; Heymans, Patrick; Trigaux, Jean-Christophe: Feature Diagrams: A Survey and a Formal Semantics. In: Proceedings of the 14th IEEE International Requirements Engineering Conference. RE '06, IEEE, Washington, DC, USA, pp. 136–145, 2006.

[SLW12] Schroeter, Julia; Lochau, Malte; Winkelmann, Tim: Multi-Perspectives on Feature Models. In: Model Driven Engineering Languages and Systems. Springer Berlin Heidelberg, 2012.

[SSA14a] Seidl, Christoph; Schaefer, Ina; Aßmann, Uwe: Capturing Variability in Space and Time with Hyper Feature Models. In: Proceedings of the 8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS). VaMoS'14, 2014.

[SSA14b] Seidl, Christoph; Schaefer, Ina; Aßmann, Uwe: DeltaEcore-A Model-Based Delta Language Generation Framework. In: Modellierung. Modellierung'14, 2014.

[SSA14c] Seidl, Christoph; Schaefer, Ina; Aßmann, Uwe: Integrated Management of Variability in Space and Time in Software Families. In: Proceedings of the 18th International Software Product Line Conference (SPLC). SPLC'14, 2014.

[Th11] Thüm, T.; Kästner, C.; Erdweg, S.; Siegmund, N.: Abstract Features in Feature Modeling. In: 15th International Software Product Line Conference (SPLC). 2011.

[vGBS01] van Gurp, J.; Bosch, J.; Svahnberg, M.: On the Notion of Variability in Software Product Lines. In: Proceedings of the Conference on Software Architecture. 2001.