V1E: A Kernel for Domain-specific Textual Variability Modelling Languages

Sobernig, Stefan; Leßenich, Olaf

Published: 01/12/2020

Document Version
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
v1e: A Kernel for Domain-specific Textual Variability Modelling Languages

Stefan Sobernig
stefan.sobernig@wu.ac.at
Institute for Information Systems and New Media,
WU Vienna
Vienna, Austria

Olaf Leßenich
olaf.lessenich@wu.ac.at
Institute for Information Systems and New Media,
WU Vienna
Vienna, Austria

ABSTRACT
v1e is a language kernel for textual variability modelling built on top of the language-development system DjDSL. As a language kernel, v1e provides a minimal but extensible set of abstractions to implement families of domain-specific languages for textual variability modelling. v1e provides for a small and versatile abstract syntax to encode feature models using multiplicity constraints and canonical semantics. v1e offers built-in analysis support, such as configuration validation, by maintaining internal BDD representations. A derived language becomes realised as a collection of extensions dependent on the language kernel. v1e is designed to be highly extensible and embeddable, e.g., as a dynamic library or as a REPL shell. In this paper, we showcase a selected derived language and the design decisions involved: a kernel implementation of TVL on top of v1e. We conclude the paper by pointing out current limitations (e.g., representing attributed variability models) and future directions (e.g., analysis support beyond BDD).

KEYWORDS
variability modelling, language kernel, language family, language product line, domain-specific modelling, modelling framework

1 INTRODUCTION
In domain engineering using product-line techniques, a domain engineer analyses the application domains of a family of software products. In a dedicated step of variability modelling, commonalities and differences between the software products are recorded. Differences are expressed as optional features. The results are documented in terms of a variability model. A variability model, therefore, depicts the number of derivable software products (variants) and their properties using a well-defined and expressive variability modelling language (e.g., a feature diagram). The variability model is then put to use for different analysis tasks, at different stages of product-line engineering, including but not limited to static product-line analyses, validation of feature selections, test planning, and code inspections [3, 5, 30].

The majority of early variability modelling languages offered (or were even limited to) a graphical notation inspired or derived from the original FODA notation. Early textual notations have been proposed for software-language engineering and for application generators; and gained momentum when it came to modelling large domains [8, 29]. As for variability-modelling languages with a primary textual concrete syntax, a number of suggestions have been put forth. ter Beek et al. [29] provide a systematic overview of approaches to textual variability modelling, ranging from early exemplars (FDL, GUIDSL) to more recent ones (Clafer, FAMILAR, Velvet, TVL). The approaches are contrasted regarding different dimensions, e.g., support for modelling in the large and the supported constraint types. Besides general-purpose variability modelling, modelling variability is frequently required by modelling languages targeting specific, but different application domains. While such domain-specific modelling languages operate on domain-specific data structures (e.g., to capture attributes or to define constraints), there is potential for reuse of basic variability abstractions, similar to the idea of a “calculation core” for domain-specific expressions [33].

In this paper, we report on our ongoing work towards v1e as a language kernel providing a minimal but extensible set of abstractions to implement families of domain-specific languages for textual variability modelling (see Section 2). From a bird’s eye perspective, v1e is unique in adopting a canonical representation (multiplicity encoding) throughout the syntax levels (concrete to abstract) down to the backend encoding (Binary Decision Diagrams, BDD). For the details on v1e’s abstract syntax, refer to Section 3.

The main contribution of v1e’s and its canonical abstract syntax is avoiding typical pitfalls of existing variability-modelling languages, such as the trade-off between a language’s succinctness for a modelling or analysis task at hand versus a lack of expressiveness (e.g., caused by missing or misfitting modelling abstractions). This is achieved by offering a language kernel that can be systematically extended to include modelling features to support further, more verbose types of variability models.

This convenient property of v1e is shared with so-called canonical variability-modelling languages, namely Varied Feature Diagrams (VFD; [27]) and Neutral Feature Diagrams (NFT; [16]). However, in contrast to these approaches, v1e comes as part of an integrated development infrastructure to create derived languages: the language-development system DjDSL [28]. v1e and derived languages can be reused for variability-aware software projects either as an integrated development library or as a REPL shell. In Section 4, the kernel-based re-implementation of the Textual Variability Language (TVL; [8]) is presented as a showcase. In Section 5, available design options and important limitations are discussed. Related work is revisited in Section 6, and Section 7 contains concluding remarks.

2 TOWARDS FAMILY-BASED APPROACHES TO VARIABILITY MODELLING

Language Kernels. There can be a middle ground between compact and verbose software languages, as the two extremes, specific to application domains such as those involving variability modelling
Language Families. A kernel and a library of extensions can be developed as a language family, e.g., a family of state-machine modelling languages [33] or a family of state-machine modelling languages [11, 35]. Engineering variable languages as language families shifts emphasis from developing and analysing a single language to developing and to analysing composable development artefacts for a language family. This ambition gave rise to approaches to language-product line engineering [17, 19–21] and their supporting multi-language development systems. Their shared goals are to minimise preplanning effort as well as, at the same time, to reuse development artefacts and language tooling in an unmodified manner.

The emphasis in this paper is on developing families of textual variability modelling languages as compositions of a language kernel (v1e) plus a library of kernel extensions at the levels of abstract syntax, context conditions, concrete syntax, and behaviour implementation. In Section 4, we illustrate an alternative implementation of the Textual Variability Language (TVL) as an extension of the kernel.

DjDSL. DjDSL [28] is a language-based and composition-based DSL development system. As a DSL development system, DjDSL allows a DSL developer to develop families of different DSL types (internal, external, and hybrid). DjDSL provides for a variable design and implementation of a DSL family across the different definition artefacts in an integrated manner (collaboration-based designs): abstract syntax, context conditions, and concrete syntaxes.

As for abstract-syntax definitions, DjDSL allows a DSL developer to structure an object-oriented abstract-syntax model (e.g., for representing feature models) into composable collaborations. A collaboration can directly represent optional features of an abstract-syntax family. At the level of variable textual syntaxes, DjDSL employs composable object grammars [32] based on an extended variant of parsing expression grammars (PEG). Section 4 demonstrates how a single object parsing-expression grammar (OPEG) definition suffices to implement a TVL core as an extension to v1e.

DjDSL plus v1e, the running examples as well as the code listings in this paper are available from a supplemental Web site as an executable tutorial.1

Figure 1: Overview of the key concepts of the v1e representation of feature models, in particular Choice and Feature. The example in the upper-left box depicts a model instantiation, with feature B being an optional (0..1) sub-feature of feature A. The decomposition and multiplicity edges are implemented by the same-named associations.

3 V1E: DESIGN AND IMPLEMENTATION
3.1 Abstract Syntax
In v1e, a feature model is represented by a structure of four concepts: Model, Choice, Feature, and Constraint. The abstract syntax is minimal, both in terms of element types and their relationships and their core semantics (multiplicities).

Models. A Model is the central container, factory, and lifetime context for model elements, in particular instantiations of Choice and Feature. A Model maintains references to a minimum of one Choice that represents the root element of the model. The root Feature, a key tenet of description of a feature model, is derived from this root Choice. Models are also the entry point to analysis operations based on the internal model representation (see Section 3.4).

Choices. A Choice is the model element that represents a number of presence options in terms of sub-features for the valid configurations of a feature model, along with a specific constraint on the group cardinality (multiplicity) of its child elements or candidates. The multiplicity is represented as a pair of an upper and a lower bound for the number of candidates to be expected to present in the valid configurations. The candidates are instantiations of Feature, i.e., sub-features. Each choice has exactly one context (or parent) Feature. The group cardinality described by a Choice encodes the type of sub-feature relationship between the parent and children features.

Tbl. 1 provides an overview of the multiplicity-based encoding of the most common (hierarchical) feature-model dependencies, as well as the corresponding Boolean encoding (further explained in Section 3.4). Whereas most feature-modelling languages have explicit abstract-syntax elements for the common mandatory, optional, inclusive-or, and exclusive-or dependencies between (sub-) features, Choice generalizes them as a unified modelling element. In addition, they are eligible for encoding cross-tree (non-hierarchical) constraints, including the required absence or negation (B..B).

1https://github.com/mrcalvin/djds
The interpretation of the abstract-syntax elements in Fig. 1 and the (i.e., each element has only one parent and the structure is free language.

Different encodings of atmost-k constraints are possible (e.g., binomial, binary [14]; binomial is currently implemented.) CAND is the set of candidate features for a given Choice, with \(|\text{CAND}| \in \mathbb{N}_{\geq 0}\).

| Optional sub-feature | 0..1 | \(|\text{CAND}| = 1\) | implication (\(\Rightarrow\)) |
|---|---|---|---|
| Mandatory sub-feature | 1..1 | \(|\text{CAND}| = 1\) | bi-implication (\(\Leftrightarrow\)) |
| Inclusive-or group of sub-features | 1..n | \(n = |\text{CAND}|, |\text{CAND}| > 1\) | disjunction (\(\lor\)) |
| Exclusive-or group of sub-features | 1..1 | \(|\text{CAND}| > 1\) | at-most-one (e.g., binomial enc.) |
| And group of sub-features | s..s | \(s \in \mathbb{N}_{>0}, s = |\text{CAND}|\) | conjunction \(\land\) |
| Absent (negated) sub-feature | 0..0 | (for constraints) | negation \(\neg\) |
| Lower bound | j..n | \(n = |\text{CAND}|, j \in \mathbb{N}_0, j \leq n\) | at-least-j (e.g., binomial enc.) |
| Upper bound | 0..k | \(k \in \mathbb{N}, 1 \leq k \leq |\text{CAND}|\) | at-most-k (e.g., binomial enc.) |
| Lower and upper bound | j..k | \(j, k \in \mathbb{N}_0, j \leq k \leq |\text{CAND}|\) | at-least-j \(\land\) at-most-k |

**Features.** Feature are those model elements that represent the (problem space) features. A Feature having owned instantiations of Choice is also referred to as a decomposition feature. Otherwise, a Feature is said to be a primary one. Primary features must be named. Decomposition features can also be unnamed, which is the case of auxiliary features as artefacts of certain model transformations. For named features, the Model is the naming scope and authority. A name assigned to a feature must be unique within the model.

**Constraints.** A (non-hierarchical) Constraint represents an expression string defined in an expression language external to v1e. Generally speaking, a constraint expression encodes presence or absence of certain Feature combinations in addition to the Choice structure of the Model. Expression operands represent (named) instantiations of Feature of the model, expression operators the presence (absence) conditions between Feature instantiations. Examples include subsets of the Object Constraint Language (OCL) expressions or standard formulas in a propositional-logic (PL) language.

When constraints are omitted, a model forms a tree structure (i.e., each element has only one parent and the structure is free of cycles). In presence of constraints, the model is described by a directed acyclic graph (DAG).

### 3.2 Semantics

The interpretation of the abstract-syntax elements in Fig. 1 and the supporting notion of configuration is as follows:

**Multiplicity Encoding of Variation Points.** A Choice represents a collection of sub-features into which a parent feature is decomposed. The cardinality of this collection (i.e., the candidates Fig. 1) in is the number of Feature instantiations contained in that collection. In Tbl. 1, the cardinality is denoted as \(|\text{CAND}|\). A Choice represents additionally a constraint on the cardinality of this collection. This constraint is referred to as the multiplicity, setting valid cardinalities of the constrained collection. A cardinality is valid provided that it is not less than the lower bound and not greater than the upper bound maintained by a Choice. Typical bounds and their interpretation in terms of feature modelling are documented

---

Figure 2: Example of an and-decomposition, in Czarnecki-Eisenecker notation; left: and-group of mandatory sub-features; right: and-group including optional sub-feature.

![Diagram](image-url)

in Tbl. 1. There are two important qualifications to multiplicities in v1e.

1. There are no unbounded multiplicities, that is, there is always a constraint on the upper bound of a Choice.
2. The upper bound of a Choice cannot be greater than the cardinality of the represented collection.

**Configuration Validation.** Based on the abstract syntax and the interpretation of multiplicities represented by a Choice, a configuration is considered valid if the following conditions hold:

1. Every element of a Model is assigned a Boolean value, which is computed according to the subsequent steps.
2. All Model-level choices evaluate to true.
3. A Choice evaluates to true if at least the lower bound and at most the upper bound of its candidates features evaluate to true.
4. A Feature evaluates to true if it is included by the configuration under evaluation and if its owned Choice instantiations evaluate to true, if any.

This generic and cascading evaluation procedure implies that all Constraint instantiations evaluate to true, whether they are represented as Choices directly or otherwise. It also follows that, to become valid, the configuration must contain the root feature. While this evaluation procedure can be implemented at a known complexity [16], v1e reformulates the evaluation of configurations into a satisfiability problem (see Section 3.4).

**And-Groups vs. And-Choices.** In an and-group of sub-features, or and-decomposition, all grouped sub-features must be present in a configuration, in which the parent feature is also present, to render the configuration valid. In (graphical) notations of feature models, they are typically identified by free-standing decomposition edges, not connected by an arc (as opposed to arcs for or- and

---

For example, a typical transformation in the problem space is turning all primary features as artefacts of certain model transformations. Examples include subsets of the Object Constraint Language (OCL) expressions or standard formulas in a propositional-logic (PL) language.
xor-groups). Such and-groups may contain both mandatory and optional sub-features. The feature model on the left in Fig. 2 indicates that B and C must be present in all configurations that A is (i.e., there is just one valid configuration: A, B, C). The feature model on the right exemplifies an and-group including an optional sub-feature D. It interprets as follows: B must be present in all configurations that A is, B can be present or absent (i.e., there are two valid configurations: A, B, D and A, B). As straightforward as their interpretation may seem, and-groups cause ambiguity in semantics (e.g., for their multiplicity encoding in presence of optional sub-features) and also notational ambiguity, e.g., when a (graphical) notation allows for multiple groups per feature.

v1e avoids any ambiguities (semantic and notational) by a separation of concerns: On the one hand, an explicit and-decomposition can be modelled using a single choice (i.e., all sub-features are mandatory) or using different choices (if there are optional sub-features involved). On the other hand, multiple groups (of whatever multiplicity class) are represented using distinct choices per group.

- Model and Feature can have multiple associated Choice instantiations. All choices must evaluate to true, for the Model or Feature to evaluate to true, subsequently.
- And-groups in terms of feature modelling translate into v1e as follows:
  - Sub-features are all mandatory: A single Choice with a multiplicity constraint limiting the cardinality (lower and upper bounds) to exactly the number of candidate features. In v1e, this is referred to as an and-choice. Fig. 3a is a v1e abstract-syntax representation of Fig. 2, LHS.
  - At least one sub-feature is optional: The mandatory sub-features are grouped by an and-choice. The optional ones by a separate choice with a lower bound of 0 and the upper bound equal to the cardinality of the subset of optional sub-features. Fig. 3c is a v1e abstract-syntax representation of Fig. 2, RHS.
- When needed for expressing a domain, or transcribing models from group-aware feature-modelling languages as frontend, v1e can contain multiple choices, including choices encoding and-groups as above, at a given decomposition level such as the root feature.

Hence, and-groups and and-choices are distinct modelling elements, with the latter capable of embedding the former, depending on the kind of sub-features (mandatory, optional). It should be noted that a single and-choice can be rewritten as a number of [1..1]-choices (see Fig. 3d), so can choices of optional sub-features be defined as separate [0..1]-choices.

### 3.3 Built-In Concrete Syntax

v1e provides multiple textual concrete syntaxes. First, direct instantiation of the abstract-syntax model is supported. Second, indirect instantiation via an internal DSL syntax is offered. Additional textual syntaxes can be added using internal or external DSL techniques (see Section 4).

Lst. 4a exhibits the internal syntax for a GraphPL example. The root keyword identifies the root feature, e.g., Graph from Fig. 4b. Internally, this is transformed into a combined structure of root choice and feature. The root is decomposed into sub-features, e.g., two optional sub-features weighted and coloured from Fig. 4b.

In v1e, an optional sub-feature corresponds to a Choice of multiplicity [0..1] with one Feature as its child element. See also Tbl. 1 for a reference. Recall that a parent feature (Graph) can carry multiple instantiations of Choices (two in Lst. 4a). Semantically, this corresponds to an implicit and-decomposition.

**Constraints.** v1e allows for defining additional constraints on the hierarchical structure of Choice and Feature instantiations, which run across the hierarchy. Such constraints can be defined in terms of an auxiliary, external textual constraint sub-language and/or extra Choice instantiations owned by the Model (i.e., at the top level, outside the root hierarchy).

Textual constraints provide a small subset of Boolean expressions, including the binary operators and and or as well as the unary operator not. These operators work on operands which represent (named) features. Compound expressions must be grouped explicitly using pairs of parentheses. The core of the constraint language is kept minimal, additional operators are modelling using the primitives (implication etc.). Syntactic sugar is provided, though. The corresponding (parsing expression) grammar is documented in Lst. 9 using an EBNF-like notation.

Consider the exemplary textual constraint in Lst. 4c. It imposes an additional validation condition on an extended GraphPL variability model (not shown). Any valid configuration including the feature MST must also include weighted, but not necessarily vice versa. This is an example of an implication, modelled as using not/or. Textual constraints cannot define new features. Only references by name to those defined as part of the hierarchy under the root are permitted.

Constraints can also be represented directly as Choice instantiations owned by a Model in addition to the root choice. Both constraint types have the same expressiveness. See Tbl. 2 for an overview of the correspondences between v1e textual constraints and choices. For instance, the textual constraint in Lst. 4c corresponds to the Choice structure in Lst. 4d, and vice versa.

Lst. 4d exhibits two noteworthy details. First, a Feature can be used in an unnamed manner (see line 2). This allows for encoding a unary operator such as not. This is one use of a Feature as an auxiliary construct. The second detail is the use of a [0..0]-choice as the multiplicity encoding of the not-operator itself (see line 3).
A variability model defined using \( v1e \) can be subjected to different predefined or developer-provided automated analysis operations. \( v1e \) offers built-in support for recoding a Model\( \text{I} \) into corresponding formulas of Boolean algebra.

A Boolean formula is recognised or generated by the following grammar:

\[
F \leftarrow X \mid '0' \mid '1' \mid '\neg' F \mid F \cdot A \mid F \mid F \cdot \tau' F \mid F \cdot \tau' F \\
\lor' F \mid F \cdot (CAND) \mid F \mid F \cdot (CAND) \mid F
\]

Operators include negation, conjunction, disjunction, implication, and bi-implication (in order of appearance above). Operands are the literals 0 and 1 as well as a range of Boolean variables denoted by \( x \). A Boolean variable takes a value out of the set \( \{0, 1\} \). The interpretation of operators follows the standard truth tables. Note, however, that in this setting, their interpretation will be normalised, beyond a succinct rewrite in terms of conjunction, disjunction, and negation only, in terms of an if-then-else normalisation (INF).

Starting from a \( v1e \) model, the following steps are performed to obtain a corresponding (non-normal) Boolean formula [5]:

1. Each primary feature maps to a same named Boolean variable.
2. Each choice maps to an operator, depending on the multiplicity set and its number of candidate sub-features (see Tbl. 1 for an overview). The current implementation uses the binomial encoding:

\[
\bigwedge_{x \in \{1, \ldots, n\}, x \in X} \neg x \\
\bigvee_{x = k+1} (CAND)
\]

To reduce the number of clauses, which is \( \binom{k+1}{k+1} \) using the binomial encoding, different encodings could be implemented [14].

3. Each textual constraint is processed as-is (given that they build on a subset of Boolean algebra and the syntactic structure allows for direct processing).

4. The overall formula is the conjunction of all sub-formulas, established by iterating the choices and any constraints.

From such a corresponding formula, \( v1e \) internally computes a Binary Decision Diagram (BDD) [6, 18]. A BDD takes the structure of a rooted, directed acyclic graph or a binary tree with shared subtrees. This property results from the node and edge sets: There is a maximum of two sink nodes labelled \( \bot \) (for false or 0) and \( \top \) (for true or 1), respectively, with an out-degree of zero. Each non-sink or branch node is labelled by a Boolean variable and maintains exactly two outgoing edges, connecting two successors. The edges or successors are called the low and high edge and successor, respectively. The low edge models the consequence of assigning the variable represented by the source branch node to 0, the high edge models the variable assignment of 1. This way, a BDD represents a model of a function that maps a Boolean formula to a resulting truth value, 0 or 1, based on a given variable assignment. An assignment is one allocation of 0 and 1 to the Boolean variables of a formula (represented by branch nodes in the corresponding BDD).

The construction of a BDD from a Boolean formula can be modelled as two subsequent steps, one of normalisation, one of reduction: First, all Boolean operator occurrences are rewritten as their if-else-then equivalents using recursive application of the Shannon expansion (assuming a previously decided fixed order of variables under expansion). This results in the if-then-else normal form (INF) of the formula. Second, the set of if-then-else sub-formula is then reduced based on identical test conditions (RHS) to obtain a close progenitor of the final BDD. Intuitively, each sub-formula of an INF formula maps to a branch or sink node of the BDD, with the low edge representing the else-branch and the high edge the then-branch of the if-construct. In this reading, a BDD models a Boolean function effectively as a decision graph [6].

The resulting BDD has convenient properties: It can be stored effectively (using beads [18, Section 7.1.4]) and allows for directly...
The Textual Variability Language (TVL) [8] is a variability modelling language with a textual concrete syntax backed by an abstract syntax and semantics derived from VFD. As a language definition itself, TVL has been repeatedly implemented using different infrastructures, e.g., as a Java library or as an external DSL implemented via ASF+SDF. As a language, TVL has also been adopted by others: µTVL is a derived language subset of TVL embedded as a language component into the Abstract Behavioural Specification (ABS) language for representing feature models [7].

TVL makes a representative application case because TVL aims at covering for a broad scope of variability modelling (feature models, constraints, attributes). At the same time, derived and embedded uses of TVL such as µTVL invite to turn TVL into a language-product line for variability modelling. Beyond concepts, implementation-wise, an implementation of TVL also exhibits important challenges (i.a., handling decompositions in the presence of optional sub-features). In the following, we highlight the selected and critical steps of implementing a TVL core in line with [7, 8] using v1e as a language kernel and DjDSL as the infrastructure to implement a TVL family. Lst. 6, LHS, visualises a small TVL model using TVL concrete syntax as a running example.

4 APPLICATIONS: AN EXTENSIBLE TVL IMPLEMENTATION

The Textual Variability Language (TVL) [8] is a variability modelling language with a textual concrete syntax backed by an abstract syntax. An object parsing-expression (OEPE) can contain extended parsing expressions in an EBNF-inspired notation to process the consumed syntactic structure (parse) into an object graph [28, Chapter 5]. This way, an OEPG definition lays out two-in-one: (a) input recognition and (b) mapping the recognised input onto objects, their fields, and non-hierarchical relationships between the mapped objects.
In this application case, the resulting object graphs consist of instantiations of the extended Model, Choice, and Feature classes (see Fig. 5). A comprehensive excerpt from this grammar definition is shown in the Appendix (see Lst. 8). The basic parsing expressions were derived from the EBNF grammar of µTVL [7, Fig. 3]. Below, we elaborate on selected grammar details which explain the realisation of the mapping definitions in Section 4.1.

**Instantiation Generators.** The parsing rules in Fig. 7 specify how the extended v1e classifiers Model and Feature are instantiated through application of the corresponding rules (5, Feature-DeclInner). For this, OPEG offers generator expressions enclosed by single grave accents (`...`). Each rule, once applied, can yield one or several instantiations of a given classifier (for example, one per alternate).

**Assignment generators.** To become useful, a parsing rule can be extended to include assignment generators. These generators mark recognised and consumed values from the processed input as values to become assigned to the properties of objects created by an instantiation generator. Lst. 7 (left) shows the example of an assignment generator for a property name of the Feature classifier. Any input recognised by applying feature FID will be assigned to the name property of the subsequently created Feature instance. The object outcome of applying the rule FDeclBody, typically one or several choices, becomes assigned to a Feature's owned property. Refer to Fig. 1 for a complete overview of relevant relationships.

Assignment generators do not necessarily apply to objects generated by the same rule, but can propagate up along the rule hierarchy of a parsing grammar, as exemplified in Lst. 7 (right). The assignment generators corresponding to lower and upper as Choice properties are defined in a subordinate parsing rule (Multipl) of the Choice instantiation generator (MPGroup). This keeps generator expressions reusable. The assignment generators for the extended optional and mandatory properties of Choice are shared between all parsing rules on decomposition groups (e.g., MPGroup, AndGroup).

**Query Generators & Multi-Valued Properties.** A query expression allows for navigating and accessing the object graph under construction. Lst. 7 (right, rule Multipl), the query generator $\texttt{current card}$ is shown, with $\texttt{current}$ referring to the object computed by the closest instantiation generator (Choice). When an asterisk is matched as part of a multiplicity, the query will be executed to obtain the cardinality of sub-features of a given Choice. $\texttt{root}$ refers to the top-level object of a given parse, i.e., an instance of Model. In addition, a query generator can refer to the parse matches of the surrounding parsing expression in a positional manner. In Lst. 7 (left), the start-symbol rule uses the so-matched feature id (FID) to set the name of the root feature explicitly.

Object parsing expressions with repetition operators allow for defining multi-valued assignments to bind value collections to multi-valued properties of objects. In Lst. 7 (right), the zero-or-more occurrences of rule GDecl translates into collecting all matches of feature declarations qualified by TVL's opt into a multi-valued assignment of the optional property of Choice. Feature declarations without opt enter the mandatory collection.

### 4.3 Integration (Analysis)

The TVL reference implementation [8] integrates with a SAT (Sat4J) as well as a CSP solver (CHOCO), the latter for numerically attributed variability models. v1e uses an internal BDD encoding of its variability model, realised with the BDD engine tclbdd. For the example in Lst. 6, v1e represents a variability model in v1e using three separate Choice nodes, as depicted in the centre of Lst. 6. The corresponding internal BDD is assembled by v1e via the operations $f \land (f \equiv a) \land (b \Rightarrow f) \land (f \Rightarrow c)$. This results in a variability model with two valid configurations: \{f, a, c\} and \{f, a, b, c\}. The validity of any configuration is independent of the presence of feature b. Based on this internal BDD representation, v1e offers additional services to the domain engineer using this TVL implementation, e.g., exporting a model to a SAT solver in CNF.

## 5 DISCUSSION

**Design Decisions.** v1e opens up a rich space of design options when deriving variability-modelling languages. As for the abstract-syntax design, v1e allows for implementing a direct extension to the v1e abstract syntax (see Fig. 1) or a separate abstract-syntax model for a derived language which then becomes transformed to a v1e instantiation. This can help to avoid complexity due to abstraction mismatches. As for representation options, Section 3.1 highlights the different options regarding and-decompositions vs. and-choices vs. separate choices.

Constraints can be implemented either using a Boolean expression language (with direct mapping onto a BDD) or using an extra structure of choice nodes using decomposition features. The latter avoids the complexity of enumerating presence or absence conditions. At the level of concrete-syntax design, a derived language may adopt a lightweight internal syntax first (e.g., using Tcl lists of lists). Using a parsing grammar, incremental syntax extensions (e.g., to add constraint or attribute syntax) may be added in support.
of abstract-syntext extensions without any preplanning effort using advanced grammar compositions [28, Chapter 5].

Limitations. One important limitation of v1e at the time of writing is the missing support for internally encoding attributed variability models for analysis. While different atomic types (Boolean, numeric), for different scopes (model, feature), can be represented by providing extensions to v1e’s concrete and abstract syntaxes, there is no integration with appropriate analysis engines yet. Another limitation is that the construction of internal BDD representations is currently not optimised for size. However, with regards to scalability, the same limitations as for other BDD backends apply. While our design does not intend to replace the BDD backend by, for example, an SMT solver, it is possible to export the Boolean formula for use in external analysis tools.

Next Steps. v1e will be extended to provide additional analysis and transformation operations (e.g., refactoring, specialization, generalization [31]). We will also include alternative Boolean encoding styles of multiplicity constraints [14] and adaptive BDD building to exploit the hierarchical nature of v1e models (variable ordering heuristics [22]).

6 RELATED WORK
A general introduction to variability and feature modelling is provided in [3, 10, 25]. A comparative overview of different variants of feature-modelling languages, including syntaxes and semantics, is provided by [26, 27].

Riebiesch et al. [23, 24] reviewed graphical feature-modelling languages at the time (e.g., FODA, Czarnecki/Eisenecker, FeatRSEB) and suggested to add syntax and semantics of group cardinalities (multiplicities) as known from UML. The resulting feature-modelling language was rebased on (groups of) optional sub-features that carry multiplicities. Additionally, non-hierarchical relationship types between features were considered (e.g., requires, refines). v1e builds on these conceptual foundations.

Schobben et al. [26, 27] defined, as a result of the critical-analytical review of feature-modelling languages, Varied Feature Diagrams (VFD). The abstract syntax of VFD itself was defined on the basis of a canonical, abstracted representation of feature models (Free Feature Diagram, FFD) providing a design space for graph types (DAG, tree), operators, (non-hierarchical) constraint types, and presence/absence of textual constraints. Using FFD, VFD are defined as trees with a single operator card (‘group cardinality’ or multiplicity). VFD was reported to be more succinct than alternatives, by requiring fewer syntax elements when transforming from VFD to an alternative model, at the same level of expressiveness. The key to these benefits, shared by v1e, is adopting a single operator (node type) representing feature groups and group cardinalities akin to Choice (see Section 3.1).

Neutral Feature Diagrams (NFT, [16]) are a derivative of VFD that restricts variability models to a tree shape. NFT assumes concrete or primary features, which have a correspondence in terms of feature implementations, to be modelled as terminal nodes of variability trees. NFT demonstrates the benefits of a canonical abstract syntax and applying useful transformations (e.g., from a DAG to a tree, non-terminals to terminals using auxiliary nodes). These transformations are also supported by v1e.

While Clafer [4] as well as v1e are designed as minimalistic languages, Clafer is not tailored towards feature modelling per se. For validation, Clafer provides built-in transformations to Alloy, SMT, and CSP. In contrast to v1e, IVML [13] aims at defining a maximally verbose variability-modelling language including a comprehensive set of modelling abstractions to cover for complex applications, such as modelling of service-platform ecosystems. These require non-Boolean attributes, QoS constraints, and versioning, which IVML delivers as built-ins, rather than as composable extensions on top of an IVML kernel.

FAMILIAR [1, 9] has been realised both as an external (Xtext) and as internal (Java/Scala) DSL to implement the domain of feature-modelling. In addition, it supports analysis operations and importing from and exporting into different representations. A key objective is the support of managing composite feature models using aggregate, merge (in different modes), slice, and diff operations. The operations are based on traversing the FAMILIAR representation of models and their internal encoding. Key differences to v1e are that FAMILIAR is not rooted in a canonical representation of variability models, whereas v1e does not yet support composition and reconstruction of variability models based on a backend or internal model representation.

TVL [8] is prominently covered in Section 4. We showed that v1e allows for defining a TVL-like frontend as an extension to v1e’s abstract syntax plus extensible concrete syntax. A key difference between TVL and v1e remains that v1e exposes the canonical representation (including multiplicity encoding) directly via its
We presented \textit{v1e}, a language kernel for textual variability modelling, which aims at offering domain engineers a sweet spot in the trade-off between succinctness and lack of expressiveness. This balance is reached by offering a highly extensible language kernel that provides a minimalistic abstract syntax to encode variability models using multiplicity constraints and canonical semantics. Deriving variability modelling languages from the \textit{v1e} kernel is facilitated by the DSL development system DjDSL. To demonstrate the expressiveness and extensibility of \textit{v1e}, we re-implemented the core of the Textual Variability Language (TVL) using \textit{v1e}. We discussed current limitations (no analysis support for attributed variability models) and looked at future work (optimisation of internal BDD representation).

7 CONCLUDING REMARKS

We presented \textit{v1e}, a language kernel for textual variability modelling, which aims at offering domain engineers a sweet spot in the trade-off between succinctness and lack of expressiveness. This balance is reached by offering a highly extensible language kernel that provides a minimalistic abstract syntax to encode variability models using multiplicity constraints and canonical semantics. Deriving variability modelling languages from the \textit{v1e} kernel is facilitated by the DSL development system DjDSL. To demonstrate the expressiveness and extensibility of \textit{v1e}, we re-implemented the core of the Textual Variability Language (TVL) using \textit{v1e}. We discussed current limitations (no analysis support for attributed variability models) and looked at future work (optimisation of internal BDD representation).

REFERENCES


S ← Model\(^\circ\) ROOT root:($root setRoot $0 \ FID)
(owned:FDeclBody)? \_ ;
FID ← <alnum>+
FDeclInner ← Feature\(^\circ\) name:FID (owned:FDeclInner)?
FDeclBody ← OBRACKET Group? Constraint* CBRACKET;
Group ← MPGroup / AndGroup / XorGroup / OrGroup;
MPGroup ← Choice\(^\circ\) GROUP Multipl OBRACKET GDecls CBRACKET;
Multipl ← OMP lower:($current card '* ' / <digit>+) SEPMP
upper:($current card '* ' / <digit>+) CMP;
GDecls ← GDecl (COMMA GDecl)*;
GDecl ← Optionals:FDeclInner / mandatories:FDeclInner;
AndGroup ← GROUP ALLOF OBRACKET FDeclOuter (COMMA FDeclOuter)* CBRACKET;
FDeclOuter ← Choice\(^\circ\) (lower:('0 OPT))? candidates:FDeclInner;
XorGroup ← Choice\(^\circ\) GROUP ONEOF OBRACKET GDecls CBRACKET;
OrGroup ← Choice\(^\circ\) GROUP upper:('someOf) OBRACKET GDecls CBRACKET;
Constraint ← Expr SCOLON / REQUIRE COLON FID SCOLON / EXCLUDE COLON FID;
Expr ← 'True' / 'False' / FID;
UnOp ← WS '!' WS;
BinOp ← WS ('||' / '&&' / '→' / '↔' / '==') WS;
void: COMMA ← WS ',' WS;
void: COLON ← WS ':' WS;
void: SCOLON ← WS ';' WS;
void: CPARENS ← WS ')' WS;
void: OBRACKET ← WS '{' WS;
void: CBRACKET ← WS '}' WS;
void: ROOT ← WS 'root' WS;
void: GROUP ← WS 'group' WS;
void: OPT ← WS 'opt' WS;
void: ALLOF ← WS 'allOf' WS;
void: ONEOF ← WS 'oneOf' WS;
void: SOMEOF ← WS 'someOf' WS;
void: REQUIRE ← WS 'require' WS;
void: EXCLUDE ← WS 'exclude' WS;
void: WS ← COMMENT / <space>*;
void: COMMENT ← '//' / '\\n' / '\\r';

Lst. 8: The excerpt from the OPEG defining a concrete syntax of a TVL kernel; see for the complete implementation.

Expression ← _ Term (_ BinaryOp _ Term)?;
Term ← NotOp? _ (Variable / (' Expression ')));
leaf: BinaryOp ← AndOp / OrOp;
AndOp ← 'and' / '&&';
OrOp ← 'or' / '||';
NotOp ← 'not' / '\\n';
Variable ← <alnum>+
void: _ ← <space>*;

Lst. 9: A parsing expression grammar (PEG) for textual constraints on v1e models