Usage Scenarios for a Common Feature Modeling Language

Thorsten Berger
Chalmers | University of Gothenburg
Sweden

Philippe Collet
Université Côte d'Azur, CNRS, I3S
France

ABSTRACT

Feature models are recognized as a de facto standard for variability modeling. Presented almost three decades ago, dozens of different variations and extensions to the original feature-modeling notation have been proposed, together with hundreds of variability management techniques building upon feature models. Unfortunately, despite several attempts to establish a unified language, there is still no emerging consensus on a feature-modeling language that is both intuitive and simple, but also expressive enough to cover a range of important usage scenarios. There is not even a documented and commonly agreed set of such scenarios.

Following an initiative among product-line engineering researchers in September 2018, we present 14 usage scenarios together with examples and requirements detailing each scenario. The scenario descriptions are the result of a systematic process, where members of the initiative authored original descriptions, which received feedback via a survey, and which we then refined and extended based on the survey results, reviewers' comments, and our own expertise. We also report the relevance of supporting each usage scenario for the language, as perceived by the initiative's members, prioritizing each scenario. We present a roadmap to build and implement a first version of the envisaged common language.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

software product lines, feature models, unified language

ACM Reference Format:
Thorsten Berger and Philippe Collet. 2019. Usage Scenarios for a Common Feature Modeling Language. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342403

1 INTRODUCTION

Feature models can arguably be seen as the most successful notation to model the common and variable characteristics of products in a software product line [11]. Proposed almost three decades ago, as part of the feature-oriented domain analyis (FODA) method [35], hundreds of variability management methods and tools have been built upon feature models. Of 91 variability management approaches introduced until 2011 [19], 33 have used feature models to specify variability information. The reported use of feature-modeling concepts in large commercial [9, 11] and open-source systems that have to manage different forms of variability, such as the Linux kernel [12, 13, 51] further witnesses their relevance. As an illustration, Fig. 1 shows an excerpt of the Linux kernel's model in a visual feature-modeling notation (explained in more detail in Sec. 2).

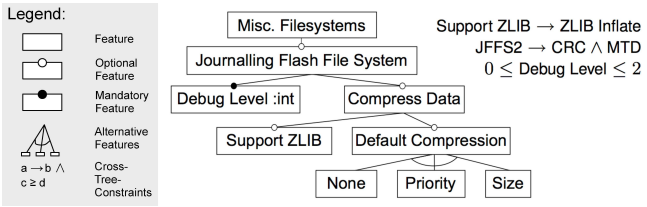

Figure 1: Feature model example (from [10])

Since the proposal of feature modeling in 1990, dozens of extensions and modifications have been proposed for feature models, often with the goal in mind to build a general feature modeling language, gradually extending the expressiveness of feature models. Examples are cardinality-based feature models [23], which support the multiple instantiation of features; attributed (a.k.a., extended) feature models [6], which allow features to have non-Boolean attributes (carrying, for instance, non-functional properties); more expressive (i.e., non-Boolean) constraint languages [46]; or even more radical approaches that combine feature and class modeling in one language [5]. Furthermore, while most academic feature-modeling notations are visual, many languages exhibiting a textual syntax have been developed for feature modeling [13, 14, 20, 27, 28].

Tooling or API-based support also emerged with the success of feature modeling in practice and research. The commercial product-line engineering tools pure::variants [17] and Gears [38], as well as the open-source tool FeatureIDE [55], are built upon feature models. In addition, many different feature-model analysis techniques and tools have been proposed [6, 42, 57]. Recent work also addressed the relative absence of processes for feature modeling by proposing modeling principles for engineers creating feature models [44].

However, despite this recognition of feature modeling in research and practice, there is still no emerging consensus on a language that would enable variability modeling in a simple and common way, while covering different possible usage scenarios. The attempt to build a standard for a common variability language, namely CVL [31], was dropped due to legal, patent-related issues. The language and its infrastructure is still available, however [58]. Establishing a standard would facilitate the interoperability of tools and would ease the sharing of feature models. Recognizing this pressuring need, a recent initiative among product-line researchers, driven by David Benavides, attempts to establish a common and simple, yet reasonably expressive feature-modeling language.

SPLC '19, September 9–13, 2019, Paris, France

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342403

This paper summarizes one of the follow-up actions that were discussed at a meeting during the Software Product Line Conference (SPLC) in September 2018 in Gothenburg. There, after a brainstorming, a set of general usage scenarios of the prospective language was elicited. After a vote, 15 main scenarios of usage were extracted, and typically two researchers, one writing, another proofreading, were assigned to detail each scenario through a common template. Each scenario has a name, a small description, an example, and some additional notes with requirements or open questions related to it. These scenarios were then described during a one-month period from mid September to mid October 2018. A survey was then built to evaluate both the clarity and the usefulness (*i.e.*, priority) of each scenario. It was distributed on the mailing list created for the initiative at the end of October 2018. Analyzing the results, we refined and extended the scenario descriptions according to the results, as well as we removed and added a few scenarios.

2 BACKGROUND

We provide an introduction into feature-modeling concepts as well as a brief recap on the history of feature-modeling languages.

2.1 Feature Models

Figure 1 shows a small feature model describing the configurable filesystem JFFS (Journalling Flash File System) in the Linux kernel. Feature models are tree-like structures of features organized in a hierarchy together with constraints among the features. While the feature hierarchy is one of the most important benefits of feature models (called ontological semantics), allowing engineers to keep an overview understanding of a product line, the primary semantics (called configuration space semantics) of feature models is to represent the valid combinations and values of features in a concrete product of a product line, restricted by constraints as follows.

In our example, the feature Debug Level is a mandatory feature (filled circle) with the value type integer; Compress Data is an optional feature (hollow circle) of type Boolean with the optional sub-features Support ZLIB and Default Compression. The latter is a feature group of type XOR, allowing to select exactly one sub-feature. Other typical kinds of feature groups are OR and MUTEX groups (not shown). These constructs express constraints, in addition to the hierarchy constraints (a sub-feature always implies its parent feature). Further constraints, named cross-tree constraints, can be expressed separately to fully capture the configuration space—shown to the right of the diagram in our example (note that ZLIB Inflate is a feature that is defined outside our excerpt of the Linux kernel model, which has grown to around 15,000 features nowadays).

2.2 Feature Modeling Extensions

The original FODA feature models have been extended in many ways. Partly inspired by a genealogy of feature-modeling successors from Kang [34], the following major extensions have been proposed.

FORM feature models [36] were introduced as part of the feature-oriented reuse method (FORM) and sub-divided models into four layers, from abstract on top to very concrete implementation-oriented features at the bottom.

FeatuRSEB feature models [30] were introduced with the FeatuRSEB methodology, aiming at an integration with use case diagrams and similar models. They are mostly equivalent to FODA models.

Hein et al. feature models [32] introduced typed relationships and binding times for features, based on industrial experience that FODA "does not provide the necessary expressiveness to represent the different types of crosslinks" in their domain. Typed relationships give rise to alternative hierarchical structures in one model, so the diagram is a directed acyclic graph, not a tree anymore.

Generative Programming feature models [21] introduced the current notation and OR groups. This notation was later extended with typed attributes and feature cardinalities [25]. Furthermore, Riebisch et al. [47] introduced arbitrary group cardinalities and constraint notations. The most significant extension were feature cardinalities [23, 24], where features, and their whole subtrees, can have more than one instance in a configuration, which has considerable impact on reasoning operations and tooling.

Clafer [5, 33] is one of the most expressive feature-modeling languages, unifying feature and class modeling. The notion of feature and that of a class is unified into a Clafer, which has a name, types, constraints, and perhaps attributes. Clafer supports multi-level modeling [18] and has a well-sepcified semantics as well as rich tooling, including instance generation, configuration, and visualization. In addition, as a textual language, it has one of the simplest and most intuitive syntaxes. Developers can use a text editor and define a feature by writing its name into a line. Hierarchy is realized by indentation (similar to Haskell and Python). Making a feature optional amounts to adding a '?' character. Feature types can also be added in a simple way.

Kconfig and CDL [13, 14] are languages to describe the variability of systems. They are developed fully independently of the research community, by practitioners who were likely not aware of the existing feature-modeling languages from researchers. Kconfig and CDL are two of the most successful languages, primarily used in the systems software domain. Kconfig [51] is used in systems such as the Linux kernel, the Busybox project, and embedded libraries (e.g., uClibc). CDL [12] is used in eCos (embedded configurable operating system). The languages in fact use concepts known from feature modeling, including Boolean, int, and string features; a hierarchy; feature groups; and cross-tree constraints. However, the languages also bring additional concepts, mainly to scale feature modeling. Specifically, they provide: visibility conditions, modularization concepts, derived defaults / derived features, and hierarchy manipulation. In addition, they provide expressive constraint languages with three-state logics for binding modes, as well as comparison, arithmetic, and string operators. Finally, all use domain-specific vocabulary, enhancing their comprehension for the developers of the systems. Details are found in Berger et al. [13, 14]. Extracted models and an infrastructure to analyze them is also available online.[1]

In addition to extensions brought by these languages (e.g., diagram shapes, layers, binding modes, expressive constraints, cardinalities, and typed edges) we find some further concepts in the literature. Among these are defaults [21, 48] and visibility conditions [26]. The latter are usually part of decision modeling languages, which share many commonalities with feature models [22].

[1] https://bitbucket.org/tberger/variability-models

Table 1: Refined scenarios descriptions: modified slightly (○) or substantially (◐) by us; or is completely new (●).

	scenario	description	example	details
Christoph Seidl	**Exchange** ◐	The language should support the bidirectional exchange of feature models between different tools. Tool vendors use the language documentation and/or existing serializers/deserializers to create important and expert functionality. Users of the tools can then leverage this functionality to export a feature model from one tool and import it in the other tool.	A feature model is created in the source tool FeatureIDE [55] and then exported into a file with the concrete syntax of the language. The file can then be imported into the FAMA framework [57] for specialized feature-model analyses.	Requirements: • The language should have a serializable concrete syntax. • The language should come with sufficient documentation about its abstract and concrete syntax to realize importers and exporters. • Ideally, serializers and deserializers are provided for the language in the form of a library (in common programming languages, especially Java) that can be used by tool vendors. • The language may be extensible and an instance may describe the level of extensions that is used. • The language may provide concepts to store tool-specific data. • Storing tool-specific data should not require changing the language or provided serializers and deserializers. Open questions: • Should tool-specific data be kept in specific language concepts or should there be tool-independent concepts to store any kind of tool-specific data? Finding a middle ground might be necessary.
Thorsten Berger	**Storage** ●	The language should allow tools to efficiently store and load feature models. Tools can use the language and its concrete syntax as the primary means to store models. Tool vendors leverage the language specification to realize fast storage and loading of models. Two subscenarios are possible: (i) the model is stored in a database, and (ii) the model is stored in a textual representation.	Consider a new product line tool that needs to store feature models. The tool vendor can develop its persistence layer by creating leveraging the language specification (i.e., the abstract syntax definition) to derive a database schema and generate CRUD functionality as well as initialize the database.	Requirements: • The language should come with an abstract syntax definition in a metamodeling notation that can be used for automated processing (e.g., generate database schemas). • The language should come with a concise and succinct [49] textual syntax. • The textual syntax should be defined in a common technology for defining concrete syntaxes, such as an ANTLR or an Xtext grammar, both of which can be used for automated processing. Open questions: • Select a language workbench (e.g., Xtext [16], MPS [15], EMF [53]) or a parser-generator technology (e.g., ANTLR [45])?
Klaus Schmid, Rick Rabiser	**Teaching and learning** ◐	The language should be easily usable for teaching. Specifically, it should be possible to describe the language within a few slides, using concepts typically taught in computer science education (e.g., types, grammars, meta-modeling). Furthermore, the language's concepts should align well with the typical and established concepts (cf. Sec. 2) that have been introduced in the product-line community and are typically taught in SPL courses (features, attributes, constraints).	The teacher describes the language with fewer than a dozen slides, and the students are able to read and write simple examples afterwards.	Requirements: • The language should have the typical visual concrete syntax of feature models. • The language should come with realistic examples (ideally extracted from real-world models, such as the Linux kernel models [13], but toy models can also be provided for simplicity, such as from SPLOT [39]. • Ideally, the language also has a concrete textual notation to illustrate how to scale models. Open questions: • Teach the textual or graphical notation? • How to keep the language simple, while being expressive? • There is a need to understand the specific examples to be provided. • Should there be different levels to be taught? (corresponding to different levels in teaching) • When teaching, can we easily relate the key concepts of the language with standard concepts taught in computer science such as requirements, components, modules (e.g., "a feature can represent a requirement")
Rick Rabiser, Philippe Collet	**Writing, reading, and editing** ○	The language should support users in writing, reading, and editing feature models in a standard text editor, targeting developers or modelers with basic programming language knowlege. Tool vendors should be able to use the language specification with automated tooling to generate an infrastructure for using the language, with typical software language engineering or transformation technology (e.g., XText, XTend or Coco/R [41]).	The user opens an editor and, given some basic knowledge about the key constructs of the language, she can instantly start writing feature models. A domain expert can easily edit feature models inside the same kind of editor. The generated language infrastructure contains a modern editor with syntax formatting, highlighting, code completion, and syntax checking.	Requirements: • The language should provide a simple and human-readable textual concrete syntax. • The language definition should be independent of a particular generation technology. • The language should allow the use of standard text editors. • Instances should be editable in standard IDEs, such as Eclipse, IntelliJ IDEA or Microsoft Visual Studio. • The language's parser should be easy to integrate in other tool chains. • Ideally, the requirements of the scenario **Domain modeling** apply.
David Benavides, José Galindo	**Model generation** ◐	Model generation (a.k.a., instance generation) automatically creates instances (models) of the language, typically aiming at instances with certain properties, such as size, coverage of language concepts, or other structural characteristics (e.g., cross-tree constraints ratio [8, 40, 50]). Tool developers can use it to generate a set of models, useful for functional testing and performance testing of the different tools supporting the language.	A tool developer launches the instance generation tool, inputs the desired properties of the model to be generated, and obtains the desired model(s).	Requirements: • The language specification (syntax and semantics) should allow for a translation of the complete semantics into a representation in a formal language. • The formal language should allow instance generation (e.g., Alloy), with instances that can be expressed in the original language's syntax (so, instantiated model in the formal lanugage should be structurally similar to the target model in the new feature-modeling language). • Ideally, the instance generation can be interactive, also showing conflicting constraints and counter-examples.

	scenario	description	example	details
Thorsten Berger	**Domain modeling** ○	The language should support early and creative software-engineering phases by allowing concept/domain modeling in terms of features. Specifically, it should allow creating features in a hierarchy, without having to specify feature value types, feature kinds (mandatory, optional), feature cross-tree relationships, or whether they belong to a feature group. The model can be gradually refined with those concepts later. Furthermore, if typed relationships are supported, hard and soft (e.g., recommends) constraints could be distinguished, the latter can be defined for each model.	A user creates an empty model and in parallel adds features (that are simply characterized by their names) and organizes them in a hierarchy. She adds cross-tree relationships if she finds it useful and quickly re-organizes the hierarchy when the domain model becomes more clear. Later, when the structure is more stable, she defines which features are mandatory, which optional, as well as she defines the other concepts.	Requirements: • The language should provide a simple and human-readable textual concrete syntax. • The language should have a concise and succinct textual syntax. • The language should rely on conventions and defaults that allow omitting the explicit instantiation of concepts (e.g., when not specific, the default feature type should be Boolean). • The textual syntax could be inspired by Clafer (cf. Sec. 2). Open questions: • Domain/concept modeling might require: multiple feature instantiation (cardinality-based feature modeling, cf. Sec. 2) as well as multi-level modeling and ontological instantiation [18]. However, supporting these concepts (as is supported by Clafer), could complicate the language. • Support typed relationships?
Christoph Seidl, Klaus Schmid	**Configuration** ◐	The language should support the configuration activities of a feature model. It should include respective constructs for selection, de-selection, un-selection of features in an feature model under configuration. Resolution of the resulting configuration space after such configuration operations should be supported by the language. The language should also support partial configuration management. The language should support default values as in product configurators. This could be directly supported in the feature model (e.g., for alternatives).	A previously created feature model is used to determine functionality of a particular variant by selecting individual features. A partial configuration of a feature model is done by several selections and unselections. The resulting set of configurations is available in the language. The absence of any resulting configuration is detected after a conflicting set of selections (e.g., with a cross-tree constraint being violated).	Requirements: • Provide adequate syntax for configuration by non-technical stakeholders. • A configuration comprises selected features but, with more elaborate language constructs (e.g., attributes), should also include value selection. • Default configurations or exemplary configurations may be sensible as suggestions (e.g., "portfolios/profiles"). • Support partial configuration. • Support by inference engine requires different types of constraints: those that can be violated temporarily and those that cannot—a typical distinction in practical languages used for configuration. Open questions: • Should the configurations be persisted within a feature model or external to it? • Is there a unique name assumption so that features can be referenced unambiguously by name? • Are configurations first-class entities in the language? • In the case of a partial configuration, should the resulting (refined) feature model be available in the language together with the set of possible configurations?
David Benavides, Mathieu Acher	**Benchmarking** ◐	The language should be designed for tool support, and several implementations are expected to be available. There should be a well-defined set of indicators to measure the performance of the most relevant operations (e.g., analysis, refactoring, configuration completion), so to be able to compare them. The benchmarking setup would allow to compare tool support execution times of these operations in isolation (e.g., without taking into account file loading or feature model parsing times when focusing on a reasoning operation).	The user loads the model with FAMA [57], Familiar [4] or Feature IDE [55] and executes the operation 'dead features,' also measuring the completion times. Then she knows which is the best tool for that operation and model. Each tool built upon the language can run the common benchmark and automatically produce an exploitable performance result.	Requirements: • Well-engineered and specified syntax and semantics of the language. • There should be an agreement on the specificatin of certain feature-model operations. • The availability of realistic models is important. Potentially, real-world models from the systems software domain can be used (cf. Sec. 2)
José Galindo	**Testing** ◐	Feature models expressed in the language should be usable as input for testing, specifically, for configuration sampling. Features and especially their constraints should be extractable in a form that allows reducing the search space for sampling techniques. Another strategy to support testing would be to express full or partial configurations (e.g., pairs of features that are critical to test).	A software engineer creates test cases that will run with configurations that are recorded in the feature model. Furthermore, when an unwanted feature interaction is detected, the engineer will record the feature pair, to be considered in future regression testing.	Requirements: • Language concepts to represent partial or complete configurations. • Ideally, consistency checking by the language infrastructure for the configuration information. • Incorporating concepts capturing further testing-relevant information (e.g., inputs for testing dedicated features) could be useful. Open questions: • Should testing-related configuration information be stored directly in the model or in a separate kind of asset?
David Benavides, Philippe Collet	**Analyses** ◐	The language can be used in automated analysis processes where the model is used as input and an analysis result is obtained. This can comprise analyses confined to the feature model [6, 29] or those that take other artifacts into account [42, 54].	Consider a Linux distribution, such as Debian. Let us assume the packages (each representing a feature) and their dependencies are described using our language (or, more realistically, are transformed from Debian's manifests into a feature model). An off-the shelf analysis, such as "dead features" can then be used to detect packages that are not selectable.	Requirements: • Community agreement on a core set (or class) of relevant analyses. • Consider different solver strategies depending on the kinds of analyses and the constructs of the language. For instance, if we allow attributes, then, specific solver capabilities are needed. • Well-specified language syntax and semantics, also with semantic abstractions into the different logical representations required by the solvers. Open questions: • Is the representation of correspondence (and maybe performance) of the solving strategies to the different constructs and extensions part of the language definition? • Should analyses be confined to the feature model or also take other asset types into account, such as the mapping to implementation assets?

scenario	description	example	details
Mapping to implementation ○ Thomas Thüm	Feature models are often not only considered in isolation. Instead, features are typically mapped to certain assets. Depending on the use case, features are mapped to requirements, architecture, design, models, source code, tests, and documentation, among others. While the actual mapping is largely independent of the feature modeling language, it should be possible to distinguish features that are supposed to be mapped to artifacts from those purely used to structure the hierarchy (e.g., to group certain features into an alternative group) or features that are not yet implemented. So, the scenario is to support developers mapping features to the implementation.	Suppose we implement a product line incrementally. That is, we have done a domain analysis in which we created a feature model and now we implement more and more of those features over time. Assume we want to derive a product or count the number of possible products before we are done with the implementation of all features. During configuration, we do not want to make decisions that do not influence the actual product. For counting, we are not interested in the total number of valid configurations, but only in those that result in distinct products.	Requirements: • A single modifier/keyword to be assigned to every feature could be sufficient (e.g., abstract/concrete as in FeatureIDE) • A well-defined mapping language might be necessary. • Avoid common limitations. For instance, a simple language rule as applied in GUIDSL, such that every feature without child features in concrete and all others are abstract, would result in unintuitive editors and overly complex feature models if a feature with child features is supposed to be mapped to artifacts. • A challenge is that this property is not be supported in many tools. Fallback could always be to mark all features as concrete during import/export (could be default). Open questions: • Should the mapping be part of the language or realized in a separate one?
Decomposition and composition ◐ Thomas Thüm	Industrial models tend to have thousands of features. Clearly, those models are created by numerous stakeholders that may even originate from different divisions or even institutions. Models can also be built according to specific separated concerns. The language should be able to compose several feature models according to different semantics (e.g., aggregation, configuration merging). It is often necessary to decompose such a large model into smaller pieces to improve the overview and facilitate collaborative development.	The Linux kernel is defined in the Kconfig [51] language but not in a single file. The knowledge is distributed over several files according to the structure of the code base. For analyses it is typically necessary to compose them all prior to feeding them into solvers. The largest known feature model (Automotive2 [37]) was developed in terms of 40 small models that have even used different modeling languages.	Requirements: • Prioritized list of composition mechanisms from the literature (e.g., aggregation, inheritance, superimposition, configuration merging). • Simple mechanism that is easy to implement. • Perhaps dedicated support for interface feature models (cf. principle MO_3 among common feature modeling principles [44]) Open questions: • Should all the composition mechanisms that have been discussed in the literature be supported in the language? • How should the language handle the fact that depending on the composition operator, it could be possible or not to add the same model several times within another model? • Is it sufficient to have support for composition in the language, whereas decomposition is up to the users?
Model weaving ◐	The language should be able to be easily integrated with other programming languages for supporting variability modeling at design and implementation levels. Several integration levels could be considered depending on the host language's capabilities and engineering needed to provide such integration in the language. A shallow form of integration could be a simple interpreter available in the host language, exchanging input and output as strings and basic types. A deeper form of integration could be an API enabling to manipulate the high-level concepts of the language in the host language (e.g., features, feature models, configurations).	Currently there are different strategies for imposing variability in other modeling formalisms such as business processes or object-oriented design. An ideal scenario would be to use our language to integrate variability in other languages, such as BPMN.	Requirements: • Depending on the depth of the integration, static design-time or dynamic run-time model weaving might need to be considered. • List of variability mechanisms from the literature. • Understanding of the effort for realizing the mechanism for different types of assets. Open questions: • Separation of concerns is an issue to consider (support developers to separate problem and solution space). • The necessary extent of embedding of information from the feature model into other assets needs to be investigated, making realistic assumptions about the actual need.
Reverse engineering and composition ● Mathieu Acher, Tewfik Ziadi	It should be possible to use the language to represent reverse-engineered or composed feature models. The former can originate from typical reverse-engineering techniques that rely on extracted variability information [2, 3, 43, 52], the latter can originate from multiple existing feature models.	Reverse-engineering examples: Synthesize a feature model from command-line parameters and the respective source-code in which they are used, from configuration files or from product comparison matrices; re-engineer web configurators [1]; extraction of feature models and reusable assets from clone & own-based systems. Composition examples: Combine multiple product lines; or use of composition and slicing operations over feature models to build a specific viewpoint on the variability.	Requirements: • The language should be sufficiently expressive to model real-world variability and configuration spaces that are reverse-engineered (these often have non-Boolean and complex constraints). • Perhaps some traceability (e.g., the artifacts from which certain constraints stem from) or debugging information (e.g., how the constraint was calculated or whether it is abstracted by weakening a constraint to make it processable). Open questions: • How to deal with constraints among features or other information extracted that is not expressible in the language (e.g., child features that exclude their parents in the Linux kernel model [13]. • As such, ground truth models may not be expressible in our language. • Empirical validation of the language: How to validate that our language is sufficient for reverse-engineering variability, especially from legacy systems? • Could round-trip engineering (which is a hard problem) be supported?

3 EMERGING USAGE SCENARIOS

As explained above (Sec. 1), the participants of the initiative's first meeting during SPLC 2018 in Gothenburg authored an original set of usage scenarios, each of which having a name, a description, an example, requirements, and potentially open questions for discussion. These were then evaluated in an online survey, created by David Benavides, targeting participants of the initiative—also those who did not attend the Gothenburg meeting, but are registered on the respective mailing list (featuremodellanguage@listas.us.es).

The survey elicited the perceived clarity of the scenario and the perceived usefulness. The original formulations of the scenarios are available in our online appendix [7], together with more detailed survey data. Specifically, using Likert-scale questions, for each scenario it asked:

(1) *What is the clarity of the scenario?* Either: 1 (not clear at all), 2 (not clear), 3 (more or less), 4 (clear), 5 (very clear).
(2) *What is the usefulness/priority of the scenario?* Either: 1 (not useful at all), 2 (not useful), 3 (more or less), 4 (useful), 5 (very useful).

The survey was distributed among the mailing list of 25 interested persons in the building of the common language at the end of October 2018 for a period of 7 weeks. 15 answers were collected.

3.1 Survey Analysis and Scenario Refinement

The results of the survey corresponded to 14 to 15 answers on each scenario, as one participant did not evaluate all scenarios. With respect to the clarity of each scenario, the results were very diverse, with some scenarios being mainly viewed as clear, while some others were not considered as clear enough. Consequently, we decided to improve the descriptions, as mentioned above (Sec. 1), to provide a better set of descriptions. Table 1 shows our final set of scenarios, refined and extended compared to those formulated after the initiative's first meeting. The table acknowledges the original authors, but we indicate whether we have modified the scenario only slightly (○), whether we did substantial changes (◐), or whether the scenario is completely anew (●). For the latter, the stated authors are also those of the new formulation.

Specifically, compared to the original set (cf. appendix [7]), we removed the scenarios **Language-Specific Characteristics** (since it was a design recommendation instead of a usage scenario or a requirement), **Storage** (since the text primarily described the sharing of models, which is covered by scenario **Exchange**), and **Translation to logics** (since it is not a prime usage scenario performed by a language user or tool developer as such, but represents a design decision and technicalities necessary to realize the majority of the other scenarios). We added the scenario **Reverse engineering and composition**, which was not formulated out by the time of the survey.

Figure 2 shows violin plots about the survey's answers with respect to the usefulness, indicating the scenario's relevance. A similar violin plot for clarity is available in our appendix [7]. Given the change, the result for **Storage** should be taken with care.

3.2 Design Recommendations

To some extent, the original usage scenario descriptions contained information about the realization of the language, which was out of the scope of this early phase of collecting usage scenarios for

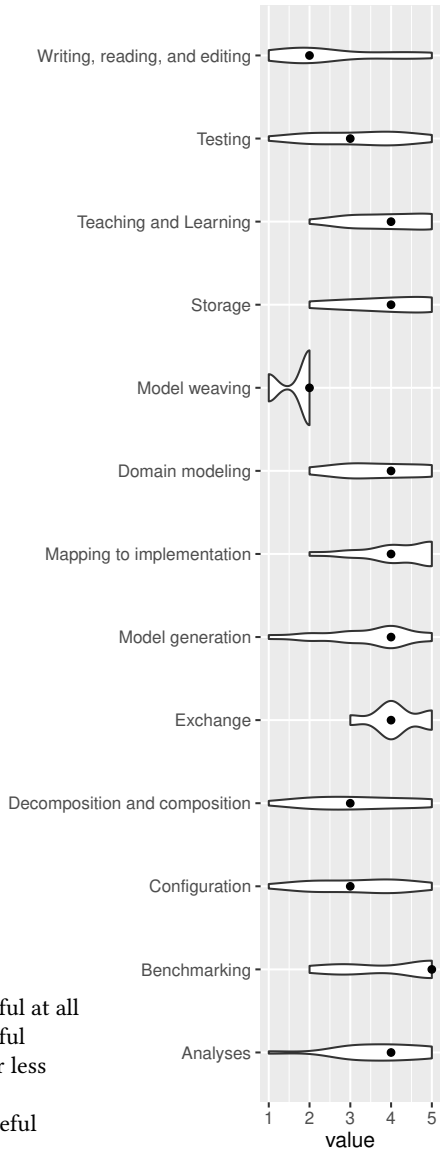

Figure 2: Perceived usefulness of the scenarios (without scenario reverse engineering and composition, cf. Sec. 3.1)

users and tool developers. The removed scenario **Translation to logics** is such a case, which provides valuable implementation recommendations by Thomas Thüm and Maurice ter Beek, further detailed in a separate paper [56]. Specifically, they argue that a key enabling technology for feature modeling is the ability to translate the semantics of feature models to a logical representation that can serve as an input to off-the-shelf constraint solvers. To this end, the expressiveness of the language should ideally align with relevant solvers, such as SAT, BDD, SMT or CSP solvers. To remain flexible, the language could offer different levels that classify the language concepts into different levels of expressiveness, each of which aligning with a specific class of solvers. Specifically, levels representing higher expressiveness allow more language concepts, but limit which solvers are applicable. For instance, if the language would

have a level that allows specifying non-Boolean feature attributes (representing, for instance, cost or performance properties) and quantitative constraints among them, this would exclude the use of solvers relying on propositional logics (e.g., SAT and BDD solvers).

To obtain language levels, Thüm and ter Beek propose mapping language concepts to levels of expressiveness, followed by identifying the priority of supporting each level based on the usage scenarios. For instance, a language level that allows quantitative modeling could support modeling (at least parts) of continuous behavior in cyber-physical systems. Thüm and ter Beek also note that language concepts might interact with respect to expressiveness, which needs to be taken into account when designing the language.

4 A PRELIMINARY ROADMAP

Our results suggest the following aspects to be considered in specifying the future language and in the workshop.

As we believe that we need to incrementally build the language features to make progress, a first set of features can be devised from the scenarios that are perceived primarily useful. According to the median value of the distribution shown in Fig. 2, this set would then comprise the following scenarios: **Exchange**, **Storage**, **Domain Modeling**, **Teaching and Learning**, **Mapping to implementation**, **Model generation**, **Benchmarking**, and **Analyses**.

From these scenarios, we can imagine some design decisions to be discussed and validated to get to a first version of the language:

- A simple textual language seems to meet the challenges from the scenario **Exchange**.
- Realization of the language's abstract and concrete syntax using a common language workbench (e.g., Eclipse EMF with Xtext) can support the scenario **Storage**.
- Incremental and partial creation of a feature model is needed for **Domain Modeling**. It directly affects the scope of what we could put inside the first set of functionalities.
- For a first set of functionalities meeting **Teaching and Learning**, simplicity of the language for writing, editing, and configuring, should be kept in mind.
- The scenarios **Model generation**, **Benchmarking**, and **Analyses** could be easy to meet in a first version if propositional feature models are chosen as a first level of expressiveness.
- *Mapping to implementation* is not an easy scenario to meet, as it is an open problem depending on the artifacts and variability realization technique.

Considering the current set of feature-modeling languages that are available, Clafer appears to meet most of the requirements. As described in Sec. 2, it is one of the most expressive languages while having a concise and succinct textual syntax, accompanied with formally defined semantics, and coming with substantial tooling. On the negative side, however, is the complexity and richness of the language, which might be problematic, but could be addressed by specifying a subset language level to accommodate certain subsets of usage scenarios.

With these aspects in mind, some open questions arise, as a basis for discussion during the workshop:

- Would the first kernel of functionalities of the language be designed and implemented at the same time? Implementation would enable to validate scenarios automatically through a continuous integration pipeline.
- Once the implementation subject is raised, the textual language implementation choices are raising at well: it could be a fluent API, an external or internal DSL, or a clever combination.
- Could the scenario **Analyses** be used with its first example, *i.e.*, running a dead-feature analysis, as a validation scenario for the implementation part of the language? Still, what analyses are useful and how their scenarios should be made clearer must be discussed. Similarly, could the **Benchmarking** scenario be also added in the same way, as its first example is a benchmark over the dead-feature computation?
- Could the language Clafer provide a reasonable basis for realizing the desired language, potentially by introducing language levels into Clafer, reducing its complexity for many scenarios?

For the workshop, we suggest to conduct a second evaluation of the refined and extended usage scenarios we presented in this paper. We plan to update our appendix [7] with the new results. The survey should again elicit clarity and usefulness, to increase our confidence in the scenarios. It should also re-open the discussion about further scenarios that need to be realized. For instance, a scenario that was briefly discussed during the first workshop was the collaborative creation of feature models, but not formulated out. Collaborative creation of feature models might be relevant for domain modeling; however, common wisdom on the processes and organizational aspects of feature modeling suggests that the distribution of the brittle variability information, and the maintenance of feature models by more than a small core group, is not feasible [10].

5 CONCLUSION

In this paper we contributed 14 usage scenarios for a simple and common feature-modeling language, to be finally established as a standard for feature modeling. We refined and extended formulations for a set of scenarios originally formulated by members of the initiative—experienced researchers from the product-line community that have some expertise in several facets of the creation and maintenance of important functionalities of such a language. We relied on a survey that was created for eliciting the clarity and relevance (i.e., usefulness or priority) of each scenario. We reported the survey results, presented the scenarios, and proposed a roadmap to support the next steps of the initiative. From these results, we observed the emergence of a smaller set of scenarios, seen as clearer and most useful, which could make a first kernel of the targeted language. We expect these insights to help in driving discussions and making decisions during the workshop.

ACKNOWLEDGMENTS

We would especially like to thank David Benavides for starting and steering this initiative, and for creating the survey questionnaire upon the initial scenario descriptions. We also thank the scenario authors Mathieu Acher, Maurice Ter Beek, David Benavides, José A. Galindo, Rick Rabiser, Klaus Schmid, Thomas Thüm, and Tewfik Ziadi; as well as we thank all the other participants of the initiative's first meeting in Gothenburg 2018.

Thorsten Berger's work is supported by Vinnova Sweden (2016-02804) and the Swedish Research Council (257822902).

REFERENCES

[1] Ebrahim Abbasi, Arnaud Hubaux, Mathieu Acher, Quentin Boucher, and Patrick Heymans. 2012. *What's in a web configurator? empirical results from 111 cases.* Technical Report P-CS-TR CONF-000001. University of Namur.

[2] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. 2011. Reverse Engineering Architectural Feature Models. In *ECSA*.

[3] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. 2012. On extracting feature models from product descriptions. In *VaMoS*.

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

[5] Kacper Bąk, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Feature and meta-models in Clafer: mixed, specialized, and coupled. In *SLE*.

[6] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636.

[7] Thorsten Berger and Philippe Collet. 2019. Survey Data on Usage Scenarios for a Common Feature Modeling Language. Technical Note. Available at http://www.cse.chalmers.se/~bergert/paper/2019-tn-fml_survey.pdf.

[8] Thorsten Berger and Jianmei Guo. 2014. Towards System Analysis with Variability Model Metrics. In *VaMoS*.

[9] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. 2014. Three Cases of Feature-Based Variability Modeling in Industry. In *MODELS*.

[10] Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wasowski, and Steven She. 2014. Variability Mechanisms in Software Ecosystems. *Information and Software Technology* 56, 11 (2014), 1520–1535.

[11] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VaMoS*.

[12] Thorsten Berger and Steven She. 2010. Formal Semantics of the CDL Language. Technical Note. http://www.cse.chalmers.se/~bergert/paper/cdl_semantics.pdf.

[13] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. on Software Engineering* 39, 12 (2013), 1611–1640.

[14] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. Variability Modeling in The Real: A Perspective from The Operating Systems Domain. In *ASE*.

[15] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of Projectional Editing: A Controlled Experiment. In *FSE*.

[16] Lorenzo Bettini. 2013. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt.

[17] Danilo Beuche. 2004. pure::variants Eclipse Plugin. (2004). User Guide. pure-systems GmbH. Available from http://web.pure-systems.com/fileadmin/downloads/pv_userguide.pdf.

[18] Victorio A. Carvalho and João Paulo A. Almeida. 2016. Toward a well-founded theory for multi-level conceptual modeling. *Software & Systems Modeling* (30 Jun 2016).

[19] Lianping Chen and Muhammad Ali Babar. 2011. A systematic review of evaluation of variability management approaches in software product lines. *Information and Software Technology* 53, 4 (2011), 344 – 362.

[20] Andreas Classen, Quentin Boucher, and Patrick Heymans. 2011. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming* 76, 12 (2011), 1130 – 1143.

[21] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, Boston, MA.

[22] K. Czarnecki, P. Gruenbacher, R. Rabiser, K. Schmid, and A. Wasowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *VaMoS*.

[23] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process Improvement and Practice* 10, 1 (2005).

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

[25] Krzysztof Czarnecki, Kasper Østerbye, and Markus Völter. 2002. Generative Programming. In *ECOOP*.

[26] Deepak Dhungana, Patrick Heymans, and Rick Rabiser. 2010. A Formal Semantics for Decision-oriented Variability Modeling with DOPLER. In *VaMoS*.

[27] Holger Eichelberger and Klaus Schmid. 2013. A Systematic Analysis of Textual Variability Modeling Languages. In *SPLC*.

[28] Holger Eichelberger and Klaus Schmid. 2015. Mapping the Design-space of Textual Variability Modeling Languages: A Refined Analysis. *Int. J. Softw. Tools Technol. Transf.* 17, 5 (Oct. 2015), 559–584.

[29] José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated Analysis of Feature Models: Quo Vadis? *Computing* 101, 5 (May 2019), 387–433.

[30] M. L. Griss, J. Favaro, and M. d' Alessandro. 1998. Integrating Feature Modeling with the RSEB. In *ICSR*.

[31] Oystein Haugen, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. CVL: common variability language. In *SPLC*.

[32] Andreas Hein, Michael Schlick, and Renato Vinga-Martins. 2000. Applying Feature Models in Industrial Settings. In *SPLC*.

[33] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. 2018. Clafer: Lightweight Modeling of Structure and Behaviour. *The Art, Science, and Engineering of Programming Journal* 3 (07/2018 2018).

[34] K.C. Kang. 2009. FODA: Twenty Years of Perspective on Feature Models. In *Keynote Address at the 13th International Software Product Line Conference (SPLC'09).*

[35] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study.* Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

[36] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.* 5 (Jan. 1998), 143–168.

[37] Sebastian Krieter. 2015. *Efficient Configuration of Large-Scale Feature Models Using Extended Implication Graphs.* Ph.D. Dissertation. University of Magdeburg.

[38] Charles W. Krueger. 2007. BigLever software gears and the 3-tiered SPL methodology. In *OOPSLA*.

[39] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: software product lines online tools. In *OOPSLA*.

[40] Marcilio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. 2009. SAT-based Analysis of Feature Models is Easy. In *SPLC*.

[41] H. Mossenbock. 1990. *Coco/R - A Generator for Fast Compiler Front Ends.* Technical Report. Johannes Kepler Universität Linz.

[42] Mukelabai Mukelabai, Damir Nesic, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *International Conference on Automated Software Engineering (ASE).*

[43] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 820–841.

[44] Damir Nešić, Jacob Krüger, Stefan Stănciulescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *ESEC/FSE*.

[45] Terence Parr. 2013. *The definitive ANTLR 4 reference.* Pragmatic Bookshelf.

[46] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. 2011. A Study of Non-Boolean Constraints in a Variability Model of an Embedded Operating System. In *FOSD*.

[47] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. 2002. Extending Feature Diagrams with UML Multiplicities. In *IDPT*.

[48] Juha Savolainen, Jan Bosch, Juha Kuusela, and Tomi Männistö. 2009. Default values for improved product line management. In *SPLC*.

[49] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *RE*.

[50] Sergio Segura, José A. Galindo, David Benavides, José Antonio Parejo, and Antonio Ruiz Cortés. 2012. BeTTy: benchmarking and testing on the automated analysis of feature models. In *VaMoS*.

[51] Steven She and Thorsten Berger. 2010. Formal Semantics of the Kconfig Language. Technical Note. http://www.cse.chalmers.se/~bergert/paper/kconfig_semantics.pdf.

[52] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse engineering feature models. In *ICSE*.

[53] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional.

[54] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (June 2014), 6:1–6:45.

[55] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85.

[56] Thomas Thüm, Christoph Seidl, and Ina Schaefer. 2019. On Language Levels for Feature Modeling Notations. In *MODEVAR*.

[57] Pablo Trinidad, David Benavides, Antonio Ruiz-Cortés, Sergio Segura, and Alberto Jimenez. 2008. Fama framework. In *SPLC*.

[58] Anatoly Vasilevskiy, Øystein Haugen, Franck Chauvel, Martin Fagereng Johansen, and Daisuke Shimbara. 2015. The BVR tool bundle to support product line engineering. In *SPLC*.