Master's Thesis

Towards a Universal Variability Language

Dominik Engelhardt

August 6, 2020

Advisors:

Prof. Dr.-Ing. Ina Schaefer

Institute of Software Engineering and Automotive Informatics

TU Braunschweig, Germany

Prof. Dr.-Ing. Thomas Thüm

Institute of Software Engineering and Programming Languages

University of Ulm, Germany

https://doi.org/10.24355/dbbs.084-202008211051-0

https://doi.org/10.24355/dbbs.084-202008211051-0

Abstract

While feature diagrams have become the de facto standard to graphically describe variability models in Software Product Line Engineering (SPLE), none of the many textual notations have gained widespread adoption. However, a common textual language would be beneficial for better collaboration and exchange between tools. The main goal of this thesis is to propose a language for this purpose, along with fundamental tool support. The language should meet the needs and preferences of the community, so it can attain acceptance and adoption, without becoming yet another variability language. Its guiding principles are simplicity, familiarity, and flexibility. These enable the language to be easy to learn and to integrate into different tools, while still being expressive enough to represent existing and future models.

We incorporate general design principles for Domain-Specific Languages (DSLs), discuss usage scenarios collected by the community, analyze existing languages, and gather feedback directly through questionnaires submitted to the community. In the initial questionnaire, the community was in disagreement on whether to use nesting or references to represent the hierarchy. Thus, we presented two proposals to be compared side by side. Of those, the community clearly prefers the one using nesting, as determined by a second questionnaire. We call that proposal the Universal Variability Language (UVL). The community awards good ratings to this language, deems it suitable for teaching and learning, and estimates that it can represent most of the existing models. Evaluations reconsidering the requirements show that it enables the relevant scenarios and can support the editing of large-scale real-world feature models, such as the Linux kernel. We provide a small default library that can be used in Java, containing a parser and a printer for the language. We integrated it into the variability tool FeatureIDE, demonstrating its utility in quickly adding support for the proposed language.

Overall, we can conclude that UVL is well-suited for a base language level of a universal textual variability language. Along with the acquired insights into the requirements for such a language, it can pose as the basis for the SPLE community to commit to a common language. As exchange and collaboration would be simplified, higher-quality research could be conducted and better tools developed, serving the whole community.

https://doi.org/10.24355/dbbs.084-202008211051-0

https://doi.org/10.24355/dbbs.084-202008211051-0

Acknowledgments

First, I would like to thank my advisors Prof. Dr. Ina Schaefer and Prof. Dr. Thomas Thüm for the opportunity to write this thesis. A special thanks to Thomas for his critical and constructive feedback. His expertise in the field of SPLE and experience in supervising theses was invaluable from the conception of the problem statement to the finishing touches.

I would like to thank the members of the SPL community for their engagement in finding a common language. Without your advancements in the field and the available tools and approaches, this thesis would not exist. I would particularly like to thank the participants of my questionnaires for dedicating the time to provide helpful comments and insights into the requirements for and the suitability of the proposed languages.

Additionally, I would like to thank the FeatureIDE team for their wonderful collaboration. Naturally, you provided the infrastructure to build upon, let me integrate the library, and did the code reviews on the pull requests.

Finally, I would like to thank Tobias Pett, Arne Sachtler, Phillipp Mevenkamp, and Lea Kubetzko for commenting on my drafts and drawing hilarious sketches.

https://doi.org/10.24355/dbbs.084-202008211051-0

https://doi.org/10.24355/dbbs.084-202008211051-0

Contents

1. Introduction **1**
- 1.1. Motivation . . . 1
- 1.2. Goals and Contribution . . . 2
- 1.3. Structure of this Thesis . . . 2

2. Background **3**
- 2.1. Software Product Lines . . . 3
  - 2.1.1. Possible Benefits of Software Product Lines . . . 3
  - 2.1.2. Dependence on the Domain . . . 4
  - 2.1.3. Feature Diagrams . . . 5
- 2.2. Domain Specific Languages . . . 6
  - 2.2.1. Benefits of DSLs Over GPLs . . . 6
  - 2.2.2. Challenges of DSLs . . . 7
  - 2.2.3. Categorization of DSLs . . . 8
  - 2.2.4. Implementation of DSLs . . . 10

3. Requirements for a Variability Language **13**
- 3.1. Guidelines for Designing DSLs . . . 13
- 3.2. Collected Requirements . . . 14
- 3.3. Analysis of Existing Languages . . . 19
  - 3.3.1. Brief Overview of Existing Languages . . . 19
  - 3.3.2. Comparison of the Presented Languages . . . 26
- 3.4. Questionnaire From the MODEVAR 2020 Workshop . . . 27
  - 3.4.1. Overview of the Questionnaire . . . 28
  - 3.4.2. Results of the Questionnaire . . . 32
- 3.5. Summary of the Requirements . . . 37

4. Proposal for a Universal Variability Language **39**
- 4.1. General Characteristics . . . 39
- 4.2. Language Concept 1: Nested Hierarchy . . . 41
- 4.3. Language Concept 2: Referenced Hierarchy . . . 42
- 4.4. Composition Mechanism . . . 43
- 4.5. Syntax of Constraints . . . 46
- 4.6. Summary of the Proposed Concepts . . . 47

5. Tool Support for UVL **49**
- 5.1. Choosing a Parser Library . . . 49
- 5.2. Grammars for the Languages . . . 51

https://doi.org/10.24355/dbbs.084-202008211051-0

5.3. UVL Parser Library 52
5.4. Integration into FeatureIDE 54

6. Evaluation **57**
6.1. Second Questionnaire 57
6.1.1. Overview of the Second Questionnaire 57
6.1.2. Results of the Second Community Questionnaire 58
6.1.3. Results of the Student Questionnaire 65
6.1.4. Summary of the Findings from the Second Questionnaire 66
6.2. Evaluating Against the Requirements 66
6.2.1. Evaluating Against the Design Guidelines 66
6.2.2. Evaluating Against the Collected Scenarios 67
6.2.3. Comparison with Existing Languages 68
6.2.4. Summary of the Evaluation Against the Requirements 69
6.3. Evaluating Scalability 70
6.3.1. Introducing Real-World Models 70
6.3.2. Editing Large Models 70
6.3.3. Storage Efficiency 73
6.4. Summary of the Evaluation 75

7. Related Work **77**

8. Conclusion and Future Work **79**

Bibliography **I**

A. Appendix **IX**
A.1. First Questionnaire IX
A.2. Second Questionnaire XII

https://doi.org/10.24355/dbbs.084-202008211051-0

List of Acronyms

API Application Programming Interface

AST Abstract Syntax Tree

CVL Common Variability Language

DSL Domain-Specific Language

EBNF Extended Backus-Naur Form

EDN Extensible Data Notation

FAMILIAR FeAture Model scrIpt Language for manIpulation and Automatic Reasoning

FDL Feature Description Language

FODA Feature-Oriented Domain Analysis

GPL General-Purpose Language

IDE Integrated Development Environment

IVML INDENICA Variability Modeling Language

JSON JavaScript Object Notation

JVM Java Virtual Machine

LSP Language Server Protocol

MOF Meta-Object Facility

MPS Meta Programming System

OCL Object Constraint Language

OpenDDL Open Data Description Language

SAT Boolean Satisfiability Problem

SMT Satisfiability Modulo Theories

SPL Software Product Line

SPLE Software Product Line Engineering

SXFM Simple XML Feature Model

https://doi.org/10.24355/dbbs.084-202008211051-0

TVL Text-based Variability Language

UML Unified Modeling Language

μTVL Micro Text-based Variability Language

UVL Universal Variability Language

VSL Variability Specification Language

XML Extensible Markup Language

YAML YAML Ain't Markup Language

https://doi.org/10.24355/dbbs.084-202008211051-0

List of Figures

2.1. Effort/costs of crafting products individually versus product-line development. 4
2.2. Example feature diagram showing a small server. 5
2.3. Process of parsing source code to an AST. 10

3.1. Voting results on the length of keywords. 32
3.2. Voting results on whether to end lines with a semicolon or break long lines. . 33
3.3. Voting results on which means of structuring to use. 33
3.4. Voting results on how to represent hierarchy. 34
3.5. Voting results on where to locate the group keyword. 34
3.6. Voting results on whether groups are expressive enough or if cardinalities or specialized constraints are required. 34
3.7. Voting results on which language features should be part of the scope of the language. 35
3.8. Voting results on the required expressiveness of constraints. 36
3.9. Voting results on what to specify in-line, by reference or in separate files. . . . 36
3.10. Voting results on whether to reuse an existing serialization format. 37
3.11. Voting results on the level of tool lock-in that is tolerable. 37

5.1. UML class diagram of the AST package. For brevity, we show the fields of the classes only, although they are private. Access is only possible through getters and setters in the JavaBeans style. 53

6.1. Voting results on which language concept is preferred (a) and whether participants would also agree to the other language, should the majority vote for that option (b). 59
6.2. Number of features of (a) typical and (b) largest feature models participants work on. 59
6.3. Number of features of typical and largest feature models participants work on. Grouped by their preferred language. 60
6.4. Voting results on how well the nesting language concept is liked before and after suggested changes. 61
6.5. Voting results on the level of agreement to the given statements about the nesting language concept. 61
6.6. Voting results on how well the referencing language concept is liked before and after suggested changes. 62
6.7. Voting results on the level of agreement to the given statements about the referencing language concept. 63

https://doi.org/10.24355/dbbs.084-202008211051-0

6.8. Voting results on how well the composition mechanism is liked before and after suggested changes. 64
6.9. Voting results on the level of agreement to the given statements about the composition mechanism. 64
6.10. Voting results on whether participants would allow multiple instantiation using the *as* keyword. 65
6.11. Box plot of the number of features per submodel for Automotive and Linux on a logarithmic scale. 72
6.12. Box plot of the maximum distance (in lines) between two siblings for Automotive and Linux models. 72
6.13. Box plots of the maximum level of indentation (tabs) found in the submodels of Automotive and Linux. 73
6.14. Comparison of the file sizes for different formats. Split into the size for storing features and constraints. 74

https://doi.org/10.24355/dbbs.084-202008211051-0

List of Tables

3.1. Overview of the collected scenarios. 19
3.2. Summary of design decisions of the presented languages regarding the concrete syntax. 27

5.1. Overview of characteristics of selected parser generators and language workbenches. 51

6.1. Summary of how the design decisions of UVL and the referencing language concept regarding the concrete syntax compare to presented languages. Decisions that are the same as in one of the concepts are accentuated with the corresponding color. 69

https://doi.org/10.24355/dbbs.084-202008211051-0

https://doi.org/10.24355/dbbs.084-202008211051-0

List of Code Listings

3.1. Server example expressed in the Feature Description Language (FDL). 20
3.2. Server example expressed in GUIDSL grammars. 21
3.3. Server example expressed in FAMILIAR. 21
3.4. Server example expressed in PyFML. 22
3.5. Server example expressed in Clafer. 22
3.6. Server example expressed in the Simple XML Feature Model (SXFM). 23
3.7. Server example expressed in the Variability Specification Language (VSL). . . 24
3.8. Server example expressed in the Text-based Variability Language (TVL). . . . 24
3.9. Server example expressed in VELVET. 25
3.10. Server example expressed in VM. 25
3.11. Server example expressed in the INDENICA Variability Modeling Language (IVML). 26

4.1. Server example represented in the nesting language concept. 42
4.2. Server example represented in the referencing language concept. 43
4.3. Example decomposition of the file system from the server example in the nesting language concept. 44
4.4. Example decomposition of the operating system from the server example in the nesting language concept. 44
4.5. Example composition of the submodels to construct the entire server. 45

5.1. Grammar for the nesting language concept in EBNF-like notation. All whitespace characters should be ignored automatically. 55
5.2. Grammar for the referencing language concept in EBNF-like notation. All whitespace characters that are not line breaks should be ignored automatically. 56

A.1. Final grammar for the Universal Variability Language (UVL) in EBNF-like notation. All whitespace characters should be ignored automatically. XIV

https://doi.org/10.24355/dbbs.084-202008211051-0

https://doi.org/10.24355/dbbs.084-202008211051-0

1. Introduction

Customers value products tailored to their needs [PBvdL05]. This is true for both hardware and software. To achieve this, many automotive companies offer a wide range of customization options for their cars. The Linux kernel is probably the most prominent example of a configurable software product. However, proper management of this variability is not a trivial task [CN01]. That is why fields of research have formed around configurable hardware and software.

Software Product Line Engineering (SPLE) is a comparatively new branch in software engineering to manage variability. It has become more popular in recent years, in both academia and industry [TAK+14]. It enables the development of variant-rich software systems in reduced time, cost, and with increased quality [PBvdL05; CN01]. Instead of entire products, individual characteristics are developed in separate components. These characteristics are often called features. From this set of features, the relevant ones for the customer are selected in a configuration step. Then the corresponding components are assembled into the finished product. These steps are ideally supported by tools allowing efficient management of the variability and automatic generation of a product from a set of features.

1.1. Motivation

For this task, there are many tools, in both industry and academia. FeatureIDE [MTS+17], pure::variants [pur20], Gears [Big20], kconfig [Zip20], AHEAD [Bat05] are just a few examples. However, all of these have different approaches, features, models, and formats since no standard exists yet. This makes it difficult to exchange models between tools.

A universal language, which everyone commits to, could be beneficial for the community in multiple ways. Communication would be easier when every user and tool uses the same language. Also, a representation in a familiar standardized language could aid in understanding an unfamiliar notation that is specialized for a specific use case in a new Software Product Line (SPL) tool. As a result, the teaching of variability and SPLE concepts could benefit from a common language and would be more consistent between universities since they could all teach the same language. Analyses that are commonly present in various SPL tools could be implemented in a separate tool, developing it in a joined effort. The common language as an interface to this tool could free developers from the need to implement analyses from other tools again to offer them in their own tool. Through comparisons and benchmarks between tools using a standard set of models, the quality and performance of the different tools might increase as well. This is because fixes and performance improvements to the common tool would benefit all tools equally.

The research community around SPLE also sees these benefits. That is why there has been a previous attempt to arrive at a common language [HWC12], but it failed due to legal reasons.

https://doi.org/10.24355/dbbs.084-202008211051-0

Recently, a new attempt has been started by the community, identifying and ranking key requirements for the language [BC19].

While we focus on SPLE in this thesis, the resulting language is more generally applicable than for modeling software-systems only. Thus, it might also be of interest in the field of knowledge-based configuration. Knowledge-based configuration is a field of study concerned with customizing a product to meet the individual needs of a customer [Stu97]. After the rise of mass production, the need for more customized products arose. This is addressed by knowledge-based configuration: trying to achieve the same level of efficiency and cost as mass-produced goods, but with highly variant products [FHB+14]. As such, the field is much older than the field of SPLE with one of the oldest approaches dating back to the early 80s [GK99]. As with SPLs, the usage of adequate technologies is required to significantly reduce the development and maintenance costs of such a system [FHB+14].

1.2. Goals and Contribution

This thesis's main goal is to provide a proposal for a universal variability language. The proposal should aid the community in agreeing on a standardized textual language for variability modeling and exchange. Either by accepting the proposal as it is or by using it and the considerations behind it as an input to a final standard.

To arrive at such a language, we consider general guidelines for designing Domain-Specific Languages (DSLs), the scenarios and requirements collected by Berger and Collet [BC19], as well as existing textual languages. Additionally, we prepare a questionnaire for the community to query their preferences and individual needs. We weigh and discuss these different sources carefully to derive concepts for a language, consisting of abstract as well as concrete syntax.

To increase usability and acceptance, we develop default tool support. It includes a parser and a printer as a library. We show the utility of the library by integrating it into FeatureIDE. By distributing another questionnaire, again targeting the research community around SPLE, and revisiting the requirements, we evaluate the language.

1.3. Structure of this Thesis

First, we give the required background information regarding variability modeling and DSLs in Chapter 2, introducing important concepts and design dimensions for variability languages. In Chapter 3, we consider general guidelines and usage scenarios, analyze existing textual languages, and present the results of the first questionnaire. We use these sources as inputs for the design decisions for the new language. In Chapter 4, we propose concepts for a new variability language. We develop tool support around the proposed language in Chapter 5. To evaluate the language we use another questionnaire to gather feedback from the community and check qualitative criteria revisiting the requirements in Chapter 6. We discuss related work in Chapter 7. In Chapter 8, we summarize our work, draw conclusions, and give an outlook on possible future work.

https://doi.org/10.24355/dbbs.084-202008211051-0

2. Background

Previously, we introduced our motivation and goals for this thesis, hinting at the possible benefits of SPLE and challenges when designing a DSL. Now we establish the relevant background for the proposed language, namely SPLs and then DSLs. In Section 2.1 we show the history, benefits, prerequisites, and the most common notation of SPLs, the feature diagram. For DSLs, we look at the motivation to design them, challenges involved, different categories and types of DSLs, implementation strategies, and generic design guidelines in Section 2.2.

2.1. Software Product Lines

In the last decades, the practice of mass production changed to mass customization, due to the need to tailor goods to the individual needs of the customers [CN01]. For instance, nowadays, customers expect to customize their new car to their specific needs. To keep the cost benefits of mass production and still offer some customization, manufacturers turned towards the approach of a common platform. Products share the platform as a common core and differ in individualized extensions which realize variable functionalities [PBvdL05].

An analogy to mass production is a common practice in software engineering. Often, software is built and tested on a well-defined computing environment (specific hardware, operating system, or framework) and then distributed to an arbitrary number of customers. However, enabling mass customization is challenging and particular techniques are needed.

One of these techniques is SPLE [PBvdL05]. Here, similar software systems share the same code base. Variable code is developed in individual modules (features). This makes them highly customizable, even though the code is developed only once instead of again for each customer. Clements and Northrop [CN01] define the term SPL as follows:

> A software product-line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

The reusable software artifacts that make up the SPL can be combined to form customized products, preferably in an automated way. These derived products are also known as *variants*.

2.1.1. Possible Benefits of Software Product Lines

The use of SPLs is promising reduced costs, improved quality, reduced time to market, and tailor-made products [ABK+13]. Figure 2.1 depicts the different relations of cost over time between traditional methods and SPLE. While there are added upfront complexity and implementation effort involved when incorporating a new product line, the cost per additional variant will be

https://doi.org/10.24355/dbbs.084-202008211051-0

smaller compared to individual implementations from scratch. Hence, a payoff is expected after delivering multiple variations of the product to different customers. Core parts of the new product do not have to be implemented again. To this end, the risk of introducing new bugs is minimized and the quality of the system is improved [CN01]. The reused artifacts are already production-proven in other variants. Thus, they have fewer bugs than unproven new software. Extensive reuse also enables reduced time to market, since a product can quickly be assembled based on the needs of the customer.

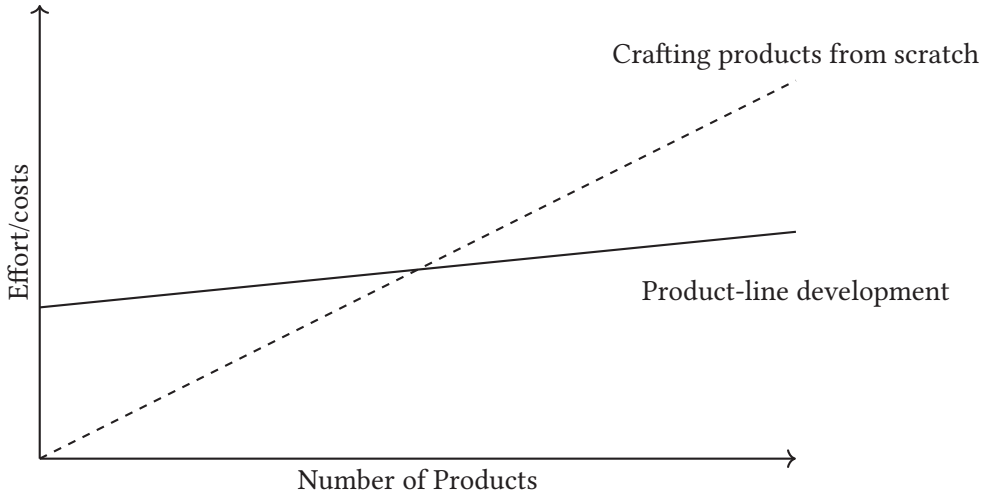


Figure 2.1.: Effort/costs of crafting products individually versus product-line development (adapted from [ABK+13]).

2.1.2. Dependence on the Domain

To achieve these benefits, one has to adhere to several constraints. Most notably the scope has to be narrowed down to a well-defined domain. Apel et al. [ABK+13] define the term *domain* as

> an area of knowledge that is scoped to maximize the satisfaction of the requirements of its stakeholders, includes a set of concepts and terminology understood by practitioners in that area, and includes the knowledge of how to build software systems (or parts of software systems) in that area.

The analysis of the domain is an integral step to understand the relevant features and their interdependencies. The term *feature* was first introduced by Kang et al. [KCH+90] as a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems", but the term is used in a variety of ways throughout the literature [ABK+13]. A more modern definition, trying to incorporate the different aspects of other definitions is provided by Apel et al. [ABK+13]:

> A *feature* is a characteristic or end-user-visible behavior of a software system. Features are used in product-line engineering to specify and communicate com-

https://doi.org/10.24355/dbbs.084-202008211051-0

monalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle.

2.1.3. Feature Diagrams

The identified features of the domain can be represented by feature diagrams, first introduced by Kang et al. [KCH+90] as part of the Feature-Oriented Domain Analysis (FODA). The feature diagram since has become the most widely used representation to specify variability [BRN+13], although there is no official specification for its syntax [SHT06]. Still, there is a certain set of characteristics that are typically supported by tools using feature diagrams.

In Figure 2.2, we give an example feature diagram as it would appear in the variability tool FeatureIDE. In the following paragraphs, we explain the typical elements based on the example.

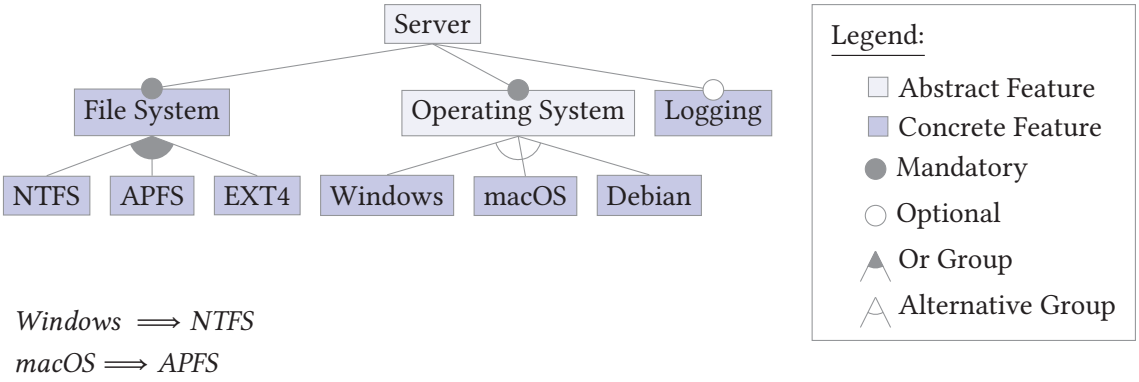


Figure 2.2.: Example feature diagram showing a small server.

Feature diagrams structure features of a configurable system in a hierarchical tree structure. A feature is shown in the example as a colored box. We distinguish between abstract and concrete features. An abstract feature does not have a corresponding implementation. This can be used for features that are only meant to be a parent of other features to clarify the structuring, simplify constraints, or to mark features that still have to be implemented. In Figure 2.2, these features are filled with a lighter shade than the concrete counterparts.

Features in the model can have other features as children. This makes the model more clear than a flat list of features. The features that are at the bottom of the tree and do not have any other children are called leaf features, while the feature at the very top is called the root feature. In most tools, only one feature can be a root node of the tree. In Figure 2.2, the feature "Server" is the root node with the child features *File System*, *Operating System*, and *Logging*. For each child feature, one can specify whether it should be optional or mandatory. Optional features have an empty circle at the top, whereas mandatory features have a filled circle. This distinction is important for the configuration of concrete systems, where optional features can but do not need to be selected. Mandatory features, on the other hand, always have to be selected for a valid configuration.

https://doi.org/10.24355/dbbs.084-202008211051-0

Another way to describe dependencies between parent and child features are feature groups. A set of child features may be contained in an *or* group or an *alternative* group. The *or* group, as the logical 'or', says that at least one of the child features has to be selected. It is denoted by a filled arc in the example. The *alternative* group, sometimes called 'xor', requires exactly one of the children to be selected for a product. The symbol for an *alternative* group is the empty arc (cf. Figure 2.2).

In addition to the hierarchical dependencies by groups and optional or mandatory features, cross-tree dependencies can be expressed by so-called cross-tree constraints. Usually, these constraints can use propositional logic, so only the features as boolean variables and basic logical operators are supported. In Figure 2.2, the constraints are listed beneath the tree. For instance, the constraint $Windows \implies NTFS$ requires that whenever the feature *Windows* is selected in a configuration, the feature *NTFS* also has to be selected for the configuration to be valid.

Since feature diagrams are a graphical notation and there is no canonical mapping to the disk, it is not a suitable exchange format. Thus, there are also several textual languages to describe feature diagrams or do variability modeling in general. We discuss these languages in Chapter 3.

2.2. Domain Specific Languages

Domain-Specific Languages (DSLs) are languages that are optimized for a specific domain. This is contrary to General-Purpose Languages (GPLs), such as common programming languages or serialization languages. Since DSLs are specific to a domain, they are (in the case of programming languages) often not Turing-complete, so it might not be possible to express every computable algorithm in the language. They are, however, more efficient than a GPL when writing a limited set of programs that are relevant to the domain. DSLs trade generality for expressiveness [MHS05].

2.2.1. Benefits of DSLs Over GPLs

GPLs have to be very general to be applicable to any problem one might encounter. The offered abstractions are often closer to the hardware and general data structures than to the concepts of the domain. Thus, programs written in a GPL can be very verbose and cluttered with so-called boilerplate code. DSLs can get rid of the unnecessary clutter. They do this by offering a language construct for each relevant concept of the domain [VBD+13]. This way the resulting code is closer to the problem it solves. To introduce a domain element one does not have to mix multiple low-level constructs from a GPL. Since there is an actual notation for each concept, programs can be shorter and more readable than their GPL counterpart.

DSLs are more restricted than GPLs. This might sound disadvantageous at first, but when the language is designed well and the restriction on the set of programs corresponds closely to the set of programs relevant to the domain, it can be an advantage. Depending on the domain it might be possible to restrict the language to only allow correct programs [VBD+13]. If this was

https://doi.org/10.24355/dbbs.084-202008211051-0

not possible, it might at least exclude whole classes of errors that can be made in GPL programs.

A restricted language with meaningful concepts and constructs is also easier to analyze than a program in a Turing-complete language [MHS05]. When an analysis is implemented for specific domain concepts, it can simply use those same concepts present in the DSL. In contrast, in a GPL the analysis first has to infer how the programmer has chosen to map the domain concepts to GPL concepts. Only then can it build an internal representation of the data using the correct abstractions to finally conduct the analysis. DSLs are also usually much smaller and simpler than GPLs [VBD+13]. That makes them easier to learn, write, and read. Often, a GPL is only understood by a few well-trained experts, while DSLs are designed to be understood by all stakeholders from a specific domain.

Since DSLs target only one specific domain, the community of users is usually much smaller and more accessible than for huge GPLs [VBD+13]. This way, the DSL can be tailored to the very specific needs of that community. It can be evolved, iterated, and customized more quickly when there is no huge community with programs that use every possible aspect of the language. Concerns of backward compatibility or complex migrations might not be as important in that case.

2.2.2. Challenges of DSLs

Of course, DSLs do not come without their own challenges. First, there is the implementation effort. A DSL has to be designed, specified, implemented in some way, and integrated into existing tools. This can be especially inconvenient since all the effort is required upfront. In contrast, when starting in a GPL, work on solving a problem can be started right away, which might feel more productive to the developer.

Another challenge for existing teams is that suddenly an additional language engineering skill is required. If no one in the team can offer such a background, it might be necessary to hire additional experts. Also, this will lead to a split in the team where one part will develop the language, whereas the other part will use it. This can always create difficulties [VBD+13].

Using a language workbench (see Section 2.2.4) or implementing editor support for the language otherwise leads to a certain level of tool lock-in, depending on the technologies used. This can be a problem when the used platform becomes unmaintained or is not flexible enough for a new feature as it can make it difficult to switch to a different technology. Then the whole language implementation would have to be repeated.

Acceptance of the language in the user base can be a further challenge [VBD+13]. When a DSL that looks similar to a programming language is proposed to users that usually do not have to write programs, this can lead to resistance. This also applies in the opposite case, when a graphical notation is imposed on programmers, as they are used to textual notations. However, even if there is no such obvious misfit, there are some people that have strong personal opinions and preferences about how a language should look. Also, introducing a new language means learning new notations and ways of solving problems, which can be a source of opposition from people reluctant to change.

Adding a DSL also results in another artifact to maintain and evolve. A language precisely

https://doi.org/10.24355/dbbs.084-202008211051-0

tailored to a certain domain will have to be adapted when the domain changes. Otherwise, the DSL might become a liability when it is suddenly difficult or even impossible to express certain domain concepts. When the used technologies change, parts of the language might have to be migrated or rewritten. Not only the DSL itself but also the knowledge of developing the DSL must be preserved. Otherwise, maintaining the DSL will be difficult, if the developers leave the team.

There is another challenge that Völter et al. call 'DSL hell' [VBD+13]. This can happen when there is enough expertise in the team to create DSLs, after having created a few. When it becomes easy to create a new DSL, developers might create a DSL for every problem they encounter, without researching first whether there might be existing languages that could be used for that problem. This results in many immature DSLs, which tackle similar problems but are still incompatible with each other.

After investing considerable resources into a language, the corresponding infrastructure, and processes, it can become difficult to change how to solve problems. When the domain changes or the previous approach is in some other way no longer applicable, it can be too much of a hurdle to try something completely new or different out of the box. To this end, the specialization and efficiency of the DSL lead to resistance against adapting to new requirements.

With these advantages and challenges at hand, it is impossible to generally say whether to use DSLs or not. This decision has to be considered carefully on a per-case basis after analyzing the domain, the requirements, and possible alternatives.

2.2.3. Categorization of DSLs

There are many kinds of DSLs. Following, we discuss a few categories in which DSLs can fall into.

Internal Versus External DSLs

When we talk about DSLs in this thesis, we usually mean external DSLs. These are stand-alone languages coming with their own syntax and tool support. Internal DSLs, in contrast, are DSLs that are embedded into another general-purpose host language [VBD+13]. This is possible in dynamic languages allowing metaprogramming. In addition to a simple API, internal DSLs aim at providing their own syntax for the added constructs, confined by the capabilities of the host language to allow custom syntaxes. As they are integrated into another language, there is usually no specific IDE support for the resulting constructs.

Language Size

DSLs differ in the number of language constructs they provide. In general, the number of supported language constructs is called the size of a language [VBD+13]. There are languages with very few constructs or keywords. Instead, they rely on powerful abstractions to provide the user with the capabilities needed. Usually, the users can define their own abstractions by composing the basic concepts. LISP is an example of a small (though not domain-specific)

https://doi.org/10.24355/dbbs.084-202008211051-0

language. Then, there are languages with many keywords and constructs for very specific things. We call these big languages.

A different approach trying to get both benefits of small and big languages by introducing additional complexity is that of modular languages [VBD+13]. Modularity, here, is not considered as the ability to express modular programs, but rather a modularity on the language level itself. A small language core can be extended by language modules adding additional syntax and IDE support. This way, one can import only those language features that are needed for the current file.

Graphical Versus Textual

When designing a DSL, one can use different syntaxes. Typically, GPLs have only one syntax that is either graphical or textual. For DSLs, it is also common to have multiple syntaxes for one language, which can be applied based on preference or concrete use case. These syntaxes can also contain both graphical and textual elements. However, usually one is either graphical or textual.

Although graphical languages have disadvantages [Pet95], some data that is inherently graphical (e.g., a floor plan of a building) can benefit from a graphical notation. Some constructs, such as hierarchical statecharts, are hard to represent in a textual notation. Also, graphical notations are considered more suitable for non-technical people and thus readily applied in various modeling languages and related DSLs. The feature diagram as seen in Figure 2.2 is one such graphical DSL.

Textual DSLs on the other hand are easier to edit using a plain text editor. There exist canonical encodings for textual languages (e.g., ASCII or UTF-8). Thus, they can be stored on disk or exchanged over the network directly, not having to be transformed into another representation in advance. Also, the theory of parsing textual languages is well understood [WSH13].

Interpreted Versus Compiled

If the language describes something executable (i.e., it has operational semantics), there are classically two ways this can be realized: compilation or interpretation. In the case of compilation, there is a program called the compiler that reads programs written in the DSL and produces a semantically equivalent program. This resulting program is composed either in another general-purpose programming language or, more commonly, directly in machine code. Compilation into machine code is usually the most efficient way to run the program. The translation into executable code, possibly including complex optimizations, has to be performed only once, while the produced executable can be run many times without overhead.

During interpretation, a program called the interpreter gets both the program expressed in the DSL and the program's input data as inputs. It then calls the appropriate functions of the program's elements directly, executing the program. This is often slower since the transformation from language concept to executable code has to be carried out at every execution. However, it is more flexible, as some information regarding the execution might only be available dynamically

https://doi.org/10.24355/dbbs.084-202008211051-0

during runtime and not at compile time. In case the language is used to describe static data only, without any executable constructs, the distinction between compiled or interpreted is not applicable.

2.2.4. Implementation of DSLs

In this section, we give an overview of how to implement an external textual DSL, as this is the kind of DSL we are developing. With implementation, we mean primarily arriving at a parser, which transforms the concrete text to a representation of the language's elements in memory.

Brief Overview of Parsing Theory

Compiler and parser theory is a huge research topic on its own. We provide a short overview of the main concepts and terminology, based on the book by Wilhelm et al. [WSH13].

The traditional way to parse a textual language is to chain scanner, parser, and name and type analysis together. This will transform the text represented in the concrete syntax to a data structure with elements of the abstract syntax, the Abstract Syntax Tree (AST). First, a scanner is used to create a token stream from the text. This token stream is passed to the parser which creates an initial version of the AST. To resolve references and types, the AST is passed to the name and type analysis, creating a decorated syntax tree. This process is visualized in Figure 2.3.

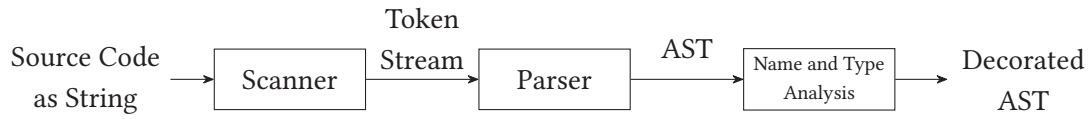


Figure 2.3.: Process of parsing source code to an AST.

The scanner, also known as the lexer, is usually not written by hand. Instead, it is specified with regular expressions, which can match inputs such as identifiers, keywords, or brackets. Based on these regular expressions, finite automata can be generated automatically. These read the text and emit corresponding tokens, whenever they reach a finite state.

The tokens are the input to the parser. The parser is specified using grammars containing production rules. Grammars can be expressed using the Extended Backus-Naur Form (EBNF). There are many types of parsers supporting different classes of grammars and being able to parse different kinds of languages. Examples include LL and LR-parsing with different lookaheads. The first 'L' in LL and LR denotes that the input is being read from left to right. The second character indicates that the parser constructs the leftmost derivation from the production rules in the case of LL while choosing the rightmost derivation for LR-parsers. We refer the reader to Wilhelm et al. [WSH13] for the complete parsing theory.

The parser emits a preliminary AST that is the input to the name and type analysis. It is the task of the name analysis to connect usages of names to their declarations, considering any scoping rules. This information is the precondition for using variables in programs. In case the language incorporates a static type system, the type analysis determines the type of every

https://doi.org/10.24355/dbbs.084-202008211051-0

program element in the AST. This information is used to ensure conformance to the typing rules (e.g., compatible types in an assignment). As in the name analysis, the generated information is attached to the elements in the AST, yielding the decorated syntax tree. This final tree can then be used for further processing steps, as all relevant contextual information is present in the tree. Examples for these steps include code generation, interpretation of programs, transformation to other languages, and more advanced analyses.

Projectional Editing as an Alternative

Projectional editing works the other way around as traditional parsing. Instead of reading text and inferring the AST, the AST is the main artifact and a textual representation of it is generated (projected), so the editing experience mimics that of textual editing. Any changes in the editor are changes to the AST first, which only after projection are reflected in the text. The AST itself is usually persisted in a structural format, such as XML. The JetBrains Meta Programming System (MPS) [Jet20] is one example of such a projectional editing approach.

The advantages of this approach include the possibility to mix graphical, textual, and tabular representations in one file and the guarantee that there will never be a representation that can not be translated into a valid AST. On the downside, editing is only possible with the projectional editor, leading to a very tight tool lock-in. Also, editing text with the indirection over the AST and back can lead to unexpected behavior compared to directly editing text.

Modeling and Language Workbenches

Language workbenches are IDEs that focus on quickly and efficiently creating DSLs and integrating them into an IDE with the usual editing support developers are used to from GPLs. The interfaces to integrate syntax highlighting, auto-completion, and other convenience features into different IDEs are highly specific to the individual IDE. Thus, language workbenches usually focus only on one IDE and offer generating these integrations from a specification that is preferably as small as possible. As a result, DSL engineering becomes quicker, more accessible to engineers without an extensive language development background, integrations become more seamless, and the overall process more feasible.

Although specifying languages is based on grammars, additional extensions are used to make the specification of editing support and overall generation of the language infrastructure easier. This leads to specifications that are not usable by other language workbenches or tools without some migration effort. Furthermore, as outlined before, the workbenches focus on one specific IDE. Thus, even though it might be possible to use a subset of the generated artifacts independently, it is at the expense of having no IDE support. To this end, the use of language workbenches leads to more tool lock-in.

Examples of language workbenches include Xtext [Ecl20], EMFText [HJK+09], and Spoofax [Met20], which focus on the Eclipse IDE, and MPS [Jet20] as a projectional language workbench generating an editor in the IntelliJ ecosystem.

https://doi.org/10.24355/dbbs.084-202008211051-0

https://doi.org/10.24355/dbbs.084-202008211051-0

3. Requirements for a Variability Language

There are many design decisions to be made when creating a DSL. Often, there are two opposite options to choose from and each decision has pros and cons for either side. For some, it can be useful to consider best practices and design guidelines. For others there is no general answer, requiring precise knowledge of the target domain, its users, and common use cases to create a sensible language. However, this precise knowledge is not readily available. As described in Section 2.1.3, there is no official specification for the syntax of feature models, the most prominent graphical notation for SPL tools. Thus, we examine four sources of information to gain this knowledge: (1) General design guidelines for DSLs, (2) collected requirements and usage scenarios by Berger and Collet [BC19], (3) existing textual languages for variability modeling, and (4) answers to a questionnaire conducted at the MODEVAR 2020 workshop. We discuss each source individually in Sections 3.1 to 3.4 and give a summary of the derived requirements for our language in Section 3.5.

3.1. Guidelines for Designing DSLs

To make our DSL design as good as possible, we list guidelines on how to best design DSLs here. These guidelines are based on the work from Karsai et al. [KKP+14]. They have experience from implementing several DSLs and DSL frameworks, as well as having conducted an extensive literature survey on the topic. The guidelines are very general, so not all of them apply to our language. In the following, we discuss the most relevant guidelines we adhered to for this thesis, grouped by the categories from the paper.

Language Purpose A language can only be of value when it fulfills the correct purpose. For this, it is important to know the aim and usage scenarios early and ask numerous questions. This we do by the end of Chapter 3, reviewing the collected usage scenarios by Berger and Collet [BC19], employing a questionnaire to the community, and discussing the results.

Language Realization Karsai et al. [KKP+14] recommend reusing existing languages, their language definitions, and their type systems as much as possible. This will help to reduce the efforts when implementing the language. This applies overall, be it an implementation from scratch or using a sophisticated language workbench. Furthermore, it will result in good practices of other languages to be reused and the resulting language will be familiar to the users of the already existing languages.

https://doi.org/10.24355/dbbs.084-202008211051-0

Language Content Which elements a language should contain is highly dependent on its purpose. Still, there are general guidelines on which elements to consider. The language should be kept as simple as possible with a limited set of elements [KKP+14]. Thus, only elements that are necessary for the envisioned usage should be included. This includes avoiding concepts that are too general or redundant to other concepts.

Concrete Syntax For the concrete syntax, Karsai et al. again encourage the reuse of existing notations for domain experts [KKP+14]. Additionally, these notations should be as descriptive as possible, making them easier to read and remember. It is also important to keep different elements distinguishable. If multiple different elements look very similar in the code and only differ in a single or a few characters, they might easily be mixed up with one another. In contrast to the previous guideline of avoiding redundancy, a little syntactic sugar (i.e., a shorter syntax for a special case of a more general construct) has the potential to improve readability. However, it should be used with caution, since too much sugar can hinder exchange and communication. This can happen when individual users rely on personal subsets of the notation to describe their problems. In this scenario, the same model might be represented in very different ways.

A balance has to be found between compactness, which is important for efficiently writing a file by hand, and comprehensibility when reading the file. However, not all the decisions have to be enforced in the grammar. The use of conventions on how to write the language should also be incorporated. An example of such a convention is to capitalize class names in Java while using lowercase for variable names, which is not enforced on a grammatical level.

According to Karsai et al. [KKP+14], semantic whitespace (i.e., layout that changes the meaning of the program) should be avoided, as it requires special caution by the developers to use consistent indentation. This is especially a concern because the different kinds of whitespace are not easily distinguishable in most editors.

Abstract Syntax For the abstract syntax, Karsai et al. recommend aligning it closely with the concrete syntax [KKP+14]. This allows for a simpler parser and printer design. Furthermore, it ensures that problems can be solved with the concrete syntax close to the concepts of the domain. Also, Karsai et al. [KKP+14] recommend to include a possibility to split documents into multiple files by providing a concept of interfaces and modularity. Organizing the files in a hierarchy using packages or namespaces and imports allows for better separation of concerns, structure, and collaboration.

3.2. Collected Requirements

In addition to these general guidelines for DSLs, we now look at more specific requirements for our language. As part of the initiative to build a common variability language by product-line researchers, Berger and Collet [BC19] collected 14 usage scenarios. First, members of the initiative submitted initial scenarios, which were then rated and ranked in a survey and finally refined to be more clear. Each member rated the scenarios for its usefulness on a scale of very

https://doi.org/10.24355/dbbs.084-202008211051-0

useful, useful, more or less useful, not useful, and not useful at all. We briefly discuss these 14 scenarios here, as they are an important source of information for the requirements of our language. We first present the scenarios that have been rated as the most useful and progress in descending order to those that were deemed less useful.

Exchange This scenario, contributed by Christoph Seidl, describes a bidirectional transfer between tools. For instance, a feature model could be created in one tool, exported into the language, then imported into another tool. There, specialized analyses could be conducted, which are not available in the first tool.

The resulting requirements for the language are that it has to be well documented and should ideally already provides a library containing a parser and a printer, so it can be easily integrated into different tools. An open question is how to deal with specific details of different formats, such as layout information, descriptions, or references to implementation artifacts. Including a construct for each possible bit of information would make the language complex and in turn harder to integrate. Excluding those would lead to a loss of information on export. A solution could be to make the language extensible with tool-specific data. This is a core concern to be considered for the language design.

Mapping to Implementation For traceability, it is desirable to have a mapping between features and other assets. These assets could be requirements, other models, implementation artifacts, tests, or documentation. Thomas Thüm formulated this scenario to consider the case when a specified product line is realized bit by bit. Then it would be useful to distinguish features that have already been implemented from those that are still missing. This way, unimplemented features can be excluded in certain analyses (e.g., when counting the number of different possible configurations). As a requirement, it should at least be possible to specify features that are not yet implemented.

This distinction can be achieved by a simple flag or keyword. In the example in Section 2.1.3, we distinguish these types of features by marking them concrete or abstract. Mappings to other assets are more complicated. The most general, but also limited, solution is to provide a reference to another file. Implementing more precise and expressive mappings to specific constructs of specific artifacts would quickly add complexity, while still being limited to a predefined set of possible targets. However, as this mapping information can be considered a special case of tool-specific data, described in the previous scenario, the same extension mechanism could be a solution instead.

Benchmarking In this scenario by David Benavides and Mathieu Acher, different tools use the language to load the same model and execute some tasks on it, so the execution time can be compared. For this, tool support for benchmarking should be added. It should be possible to load realistic real-world models into different tools, execute an operation, and measure the elapsed time.

It is unclear, how tool support for benchmarking is within the scope of the language. The

https://doi.org/10.24355/dbbs.084-202008211051-0

language and any tool support have to be designed, so that tools can easily integrate it, possibly also offering an API to create benchmark reports.

Teaching and Learning According to Klaus Schmid and Rick Rabiser, the language should be presentable with a few slides. It should utilize familiar concepts from computer science and product-line engineering, so students can already use the language after a short introduction.

Schmid and Rabiser list a concrete visual notation that is similar to feature diagrams in addition to a textual syntax as a requirement. We argue that there is currently no need for another visual notation, as the feature diagram is already considered the de facto standard [BC19; ACL+13; CBH11]. When a textual notation is agreed upon and established in the community, it might be helpful to define a formal mapping between it and the feature diagram. This way, a visual notation can be used when teaching, while at the same time providing means to unify existing approaches that use feature models or similar graphical notations.

Storage This scenario is contributed by Thorsten Berger. Efficient storage and retrieval of models should be available for variability tools. This might either be achieved using a textual syntax or by storing the model in a database. For databases, a schema should be generated automatically from the language definition.

We, however, focus on a textual syntax for storage, as it is easier to exchange and supported on any platform. Also, tools exist to support versioning, calculating differences between versions, or compression of plain text files.

Domain Modeling According to Thorsten Berger, the language should support the incremental development of models. Using a hierarchy, the model can be made more detailed over time. It should not force the user to make all the design decisions regarding group membership, cross-tree constraints, etc., upfront, but also allow for later refinement.

As feature models are already hierarchical, this scenario is naturally supported by any language that adheres to this structure, unless it imposes additional constraints on leaf features. These could change to non-leaf features through refinement and might have to be adapted. Group membership can be changed quickly by modifying the corresponding keyword.

Analyses David Benavides and Philippe Collet suggest that it should be possible to run analyses on the model. An example of such an analysis is given as identifying dead features.

We argue that to fulfill this requirement, there does not have to be specific analysis tooling for the language itself. Tools that already allow the envisioned analyses exist, so as soon as integrations into these tools are available, their analysis capabilities can be reused. However, the expressiveness of the language has a great impact on analyzability. The more expressive a language is, the more difficult it is to reduce specific analyses to problems for which efficient solvers exist (e.g., the satisfiability problem). This is an important consideration for the language design.

https://doi.org/10.24355/dbbs.084-202008211051-0

Model Generation The language should allow for the generation of random model instances for testing or benchmarking. These generated models should have predefined properties (e.g., the size of the model). The scenario is specified by David Benavides and José Galindo. One strategy to implement this might be to create a mapping to a formal language that already has tool support for instance generation.

Though it seems to be useful to be able to generate random model instances, this hardly has an impact on the design of the language. Random instances can be generated for any language, given proper tool support.

Configuration This scenario is described by Christoph Seidl and Klaus Schmid. The language should support the configuration activities for created models. Default selections, describing features that should be preselected when creating a new configuration, could be defined in the model.

An open question is, whether configurations should be persisted within the language itself or as an external artifact. An argument for mixing specification and configuration in a single file could be that it might be easier to evolve the feature model keeping the configurations consistent. However, as configurations are concrete instances of feature models, which are conceptually very different from the model itself, we favor separating them into external files. Therefore, this scenario does not have a great impact on the design of the language itself.

Testing Sampling of different configurations to test for feature interactions should be supported, as stated by José Galindo. For this, features and constraints stored in the language should be the source of the input data. It is unclear if further testing-related information is to be stored within the model or in a separate asset.

Apart from testing-related data that might be stored in the model, this scenario has no further implications for the language design, except for the need to store features and constraints. Thus, it does not have to be considered further when designing the language.

Writing, Reading, and Editing Rick Rabiser and Philippe Collet propose that the language should be editable in a simple text editor. Editors with auto-completion and other convenience features should be easy to generate for tool vendors. A challenge would be to achieve this independently of a specific generation technology.

This scenario concerns mostly the tool support generated for the language and not the language itself. Hence, we do not need to consider it for the language design.

Decomposition and Composition For very large models with thousands of features it is impractical to have all the information in a single file. Thus, the language should support some composition mechanism to still have all the necessary information available for analyses. It should be a simple mechanism that is easy to implement for tool vendors. The scenario is supplied by Thomas Thüm.

https://doi.org/10.24355/dbbs.084-202008211051-0

For teaching and researching new methods with very small example models, this scenario is not relevant. Real-world models, on the other hand, tend to consist of thousands of features, making some mechanism for better scalability necessary. Composition could be such a mechanism.

Model Weaving As proposed by Thomas Thüm, integrations into programming languages to enable variability at the implementation level should be available. Examples include business processes or object-oriented design. This scenario has been consistently ranked as not useful or not useful at all. That is why we do not give it much consideration for our language design.

Reverse Engineering and Composition It should be possible to iteratively construct (reverse-engineer) a variability model in the language from existing systems. Also, the composition of existing models to derive a new one should be supported.

This last scenario has been added by Mathieu Acher and Tewfik Ziadi after the survey was conducted, so there is no ranking available from the community. Although this scenario presents a different angle, the resulting requirements seem very similar to previous scenarios. The domain modeling scenario also describes the iterative specification of models. The composition of existing models is covered by the scenario on decomposition and composition above.

Summary of the Collected Requirements

The community has collected and ranked very different scenarios. Each scenario imposes its own set of requirements. Some of these conflict with each other. The scenario asking for efficient storage, for instance, might compromise teaching and learning, as well as writing, reading, and editing the file in a text editor. Efficient storage requires a small file size without many additional characters. In contrast, teaching and learning are likely to benefit from longer, more readable keywords. While the community agrees on the importance of some scenarios, ranking them consistently as either useful or not useful, they disagree on the usefulness of others. For example, the scenarios on configuration, testing, and decomposition are ranked useful by one part of the community, whereas an equal part considers it not useful. We focus primarily on the scenarios that were ranked most useful consistently by the community and that are relevant for the language design. These include the scenarios on exchange and efficient storage, which result in similar requirements. Further, we consider the aspects of teaching and learning and mapping to implementation, which are also ranked highly. Finally, we consider decomposition and composition. Though its usefulness is disputed in the community, we argue that it is an essential mechanism to enable scalability for real-world models. In Table 3.1, we give an overview of the usefulness and level of agreement, as ranked by the community, the relevance for the design of the language, and whether we consider it further.

A general design recommendation given by Berger and Collet [BC19] that is based on a scenario by Thomas Thüm, they later removed, is to offer different language levels. These levels could be aligned with the different solver classes (Boolean Satisfiability Problem (SAT), Satisfiability Modulo Theories (SMT), …). This way, a simple language could be used for teaching

https://doi.org/10.24355/dbbs.084-202008211051-0

Scenario	Usefulness	Agreement	Relevance for Design	Considered
Exchange	useful	strong	relevant	✓
Mapping to Implementation	useful	varied	relevant	✓
Benchmarking	useful	varied	not relevant	✗
Teaching and Learning	useful	varied	relevant	✓
Storage	useful	varied	some	✓
Domain modeling	midway	varied	not relevant	✗
Analyses	midway	varied	not relevant	✗
Model generation	midway	varied	not relevant	✗
Configuration	midway	disputed	depends	✗
Testing	midway	disputed	not relevant	✗
Decomposition and composition	midway	disputed	relevant	✓
Writing, Reading, and Editing	not useful	varied	relevant	✗
Model weaving	not useful	strong	relevant	✗
Reverse engineering and composition	–	–	not relevant	✗

Table 3.1.: Overview of the scenarios and how they meet the criteria used to determine whether they are considered for the language design. Usefulness and agreement refer to the ranking of the community. Relevance for design states whether the scenarios result in actionable requirements for the language itself.

and most projects. For more involved projects with more complex relationships between features, a language with more sophisticated language features and reduced analyzability capabilities could be used. For instance, propositional logic may be included in the basic language level, while support for first-order logic may be included in higher language levels. This recommendation is further detailed in a separate paper by Thüm et al. [TSS19].

3.3. Analysis of Existing Languages

In this section, we examine existing textual variability languages, which have been proposed in the community, and compare their benefits and drawbacks. First, we visualize the different syntaxes and features of the languages by using the server example from Figure 2.2. Thereafter, we compare their design decisions.

3.3.1. Brief Overview of Existing Languages

For each existing variability language, we give an overview of the main characteristics, discuss possible advantages and drawbacks, and show a short code snippet to demonstrate the concrete syntax. For this, we use the feature model from Figure 2.2. However, not all of the languages can represent the entire information present in the feature model. Most notably almost no language

https://doi.org/10.24355/dbbs.084-202008211051-0

supports abstract features. Also, for some languages, the constraints have to be transformed, because they do not support implications. We try to represent the example as accurately as possible in each case, to keep the code snippets comparable.

Textual Languages With References for Hierarchy

The most visually noticeable difference between the languages is whether they use references or nesting to represent the hierarchy of the features. In this section, we show those that use references and consider the others afterward. With referencing, the structure is expressed using references from the parent feature to the child features. Thus, not only trees but also more general graphs can be represented. However, for the representation of feature diagrams, only simple trees are used.

FDL First, we use the Feature Description Language (FDL) [vDK02] to express the server example, which we show in Listing 3.1. Every line defines a feature with a keyword for the group type and the corresponding children. If a child is a leaf node, it does not have to be defined in a dedicated line. Referencing it from a parent feature will also define it. The constraints are listed separately at the bottom. Note that the grammar of this language distinguishes leaf features from other features. Names of leaf features start with a lowercase letter, while other features use an uppercase letter. This restriction makes it easy to tell from the features name alone whether it is a leaf or not. However, it also hinders incremental development and refinement of models, as adding children to a previous leaf feature suddenly requires renaming it to start with an uppercase letter. In turn, the reference in the tree, as well as any cross-tree constraints have to be updated.

```
1 Server: all (FileSystem, OperatingSystem, logging?)
2 FileSystem: more-of (ntfs, apfs, ext4)
3 OperatingSystem: one-of (windows, macOS, debian)
4
5 windows requires ntfs
6 macOS requires apfs
```

Listing 3.1: Server example expressed in the Feature Description Language (FDL).

GUIDSL Grammars The GUIDSL grammars [Bat05] are structured similarly to FDL. However, instead of keywords for the different group types, they use symbols, such as '|', '+', or '[]', commonly used when specifying grammars, to describe the tree structure. Listing 3.2 depicts the server example expressed in GUIDSL. We observe that the *or* group's specification in GUIDSL is split between both the direct parent of the children and its parent, indicated with a '+' sign. This should be familiar to language engineers working extensively with grammars, but might be confusing for other users.

https://doi.org/10.24355/dbbs.084-202008211051-0

```
Server: FileSystem+ OperatingSystem [Logging];
FileSystem: NTFS | APFS | EXT4;
OperatingSystem: Windows | macOS | Debian;

Windows implies NTFS;
macOS implies APFS;
```

Listing 3.2: Server example expressed in GUIDSL grammars.

FAMILIAR The FeAture Model scrIpt Language for manIpulation and Automatic Reasoning (FAMILIAR) [ACL+13] again uses references to represent the hierarchy and symbols similar to the GUIDSL grammars. In contrast to them, *or* groups are specified only on the parent level (see Listing 3.3). The constraints in FAMILIAR neither support implications, nor a 'requires' keyword. Therefore, the constraints from our example have to be transformed into more basic boolean operators. This can be less readable than a 'requires' keyword for simple constraints, especially as the pipe operator ('|') is overloaded here. In constraints, it denotes an *or*, whereas in feature declarations it separates children into an alternative group.

```
FM (
    Server : FileSystem OperatingSystem [Logging];
    FileSystem : (NTFS | APFS | EXT4)+;
    OperatingSystem : (Windows | macOS | Debian);

    (!Windows | NTFS);
    (!macOS | APFS);
)
```

Listing 3.3: Server example expressed in FAMILIAR.

Textual Languages With Nesting for Hierarchy

The following languages show the other possibility used to denote the hierarchy. With nesting, child features are located in blocks, which are surrounded by delimiters (usually some kind of brackets) or indented by whitespace. This way, the tree structure is clearly visible for small models, but it is less compact than referencing and might not scale as well for huge models. An aspect impairing scalability is that the required indentation level might consume considerable storage space compared with the actual data. Also, the information which elements are siblings in the tree might be difficult to grasp without an editor that supports the folding of subtrees.

PyFML PyFML [Azz18] is a textual variability language based on various Python technologies, hence its name. It uses an array-style approach to specify children. Otherwise, the keywords are very similar to FDL. See Listing 3.4 for the corresponding code snippet.

We observe that specifying the hierarchy in this way, with nesting and brackets, is less compact than in the previous languages, which use references to specify the hierarchy. This

https://doi.org/10.24355/dbbs.084-202008211051-0

is especially pronounced when using a layout with meaningful indentation and one line per feature. However, in these small models, as a result, the hierarchy is clearly visible through the indentation, without the need to trace references through the model.

```
FM = Server (1..1): all [
        FileSystem (1..1): moreof [
            NTFS (0..1),
            APFS (0..1),
            EXT4 (0..1)
        ],
        OperatingSystem (1..1): oneof [
            Windows (0..1),
            macOS (0..1),
            Debian (0..1)
        ],
        Logging (0..1)
    ];
Windows implies NTFS;
macOS implies APFS;
```

Listing 3.4: Server example expressed in PyFML.

Clafer Another textual variability language is Clafer [BCW11]. It uses nesting and indentation to express the hierarchy and omits keywords where it can. This makes it look similar to code written in Python. In Clafer, constraints are not specified in a separate list at the end of the file but are children of individual constraints. See Listing 3.5 for the example in Clafer. Another design choice is that the group type is located at the parent, but the keyword is placed in front of the name of the feature. This might give the impression that the feature *is* a group and not just *has* a group.

```
Server
    or FileSystem
        NTFS
        APFS
        EXT4
    xor OperatingSystem
        Windows
        macOS
        Debian
        [Windows => NTFS;]
        [macOS => APFS;]
    Logging?
```

Listing 3.5: Server example expressed in Clafer.

https://doi.org/10.24355/dbbs.084-202008211051-0

SXFM The Simple XML Feature Model (SXFM) [MBC09] uses XML as its overall structure while incorporating a special DSL inside the tags. See Listing 3.6 for the example. The actual DSL uses nesting and is sensitive to indentation. It has very short keywords comprised of a colon and a single character. SXFM requires IDs in addition to the name for every feature which then can be used in the constraint part. It has the groups (specified with cardinalities) located between the parent and the children. SXFM does not support implications, so the constraints have to be transformed as in FAMILIAR.

```
<feature_model name="FM"><feature_tree>
    :r Server (id_srv)
        :m FileSystem (id_fs)
            :g [1,*]
                : NTFS (id_ntfs)
                : APFS (id_apfs)
                : EXT4 (id_e4)
        :m OperatingSystem (id_os)
            :g [1,1]
                : Windows (id_win)
                : macOS (id_mac)
                : Debian (id_deb)
        :o Logging (id_log)
</feature_tree><constraints>
    c1: ~id_win or id_ntfs
    c2: ~id_mac or id_apfs
</constraints></feature_model>
```

Listing 3.6: Server example expressed in the Simple XML Feature Model (SXFM).

It is unclear, how the overall XML structure is an advantage when the actual data is modeled in its separate DSL. It seems as if one would require an XML library in addition to the parser for the DSL. The single-character keywords might be fast to type but impede readability. Further, the advantage of being fast to type is hampered by the requirement for unique identifiers in addition to feature names. This way, every feature name has to be typed twice, unless shorter unique IDs, such as incrementing numbers, are used. However, that would considerably decrease the readability of constraints relying on those IDs for their references.

VSL Another textual approach to variability modeling is the Variability Specification Language (VSL) [APS+10]. Instead of indentation to express the extent of blocks, it uses parentheses. VSL also uses cardinalities instead of predefined groups that are located between parent and children. Although cardinalities are more expressive than predefined groups, such as *or* and *alternative*, they can also be more difficult to read at a glance. The syntax for constraints is also very different from other approaches before with the implication written as `link a = needs => b;`. This might be unintuitive for users that are familiar with the mathematical notation or the simplified constraints using keywords such as `requires` or `excludes`.

https://doi.org/10.24355/dbbs.084-202008211051-0

```
featureModel FM {
    Server! (
        FileSystem! ( [1..*] (
            NTFS, APFS, EXT4
        )),
        OperatingSystem! ( [1] (
            Windows, macOS, Debian
        )),
        Logging?
    );
    link Windows = needs => NTFS;
    link macOS = needs => APFS;
}
```

Listing 3.7: Server example expressed in the Variability Specification Language (VSL).

TVL The Text-based Variability Language (TVL) [CBH11] uses full-length keywords, curly braces, and nesting to represent feature models (Listing 3.8). Constraints in TVL are not specified in a separate list at the end of the file but are children of other features. A convention is to place a constraint beneath one of the referenced features in the constraint. This enables better scalability for very large models with many hundred constraints, because the constraints are located near the involved features, instead of hidden in a long list of constraints. However, for small models, users might miss the overview of all constraints as a single list at the end of the file and would need tool support to provide it.

```
root Server {
    group allOf {
        FileSystem {
            group someOf {NTFS, APFS, EXT4}
        },
        OperatingSystem {
            group oneOf {Windows, macOS, Debian}
            Windows requires NTFS;
            macOS requires APFS;
        },
        opt Logging
    }
}
```

Listing 3.8: Server example expressed in the Text-based Variability Language (TVL).

μTVL The Micro Text-based Variability Language (μTVL) [CMP+10] is based on TVL but aims to be simpler. The sole difference visible in our example is that simple 'requires'-constraints can be specified in a shorter form. The parent feature can be omitted when it is on the left side of the constraint. Thus, we do not include a dedicated listing for this language.

https://doi.org/10.24355/dbbs.084-202008211051-0

VELVET VELVET [RST+11], see Listing 3.9, is similar to TVL but uses more keywords. At first, it looks similar to a Java program with curly braces and semicolons, although it is not executable, merely describing feature models. With the most keywords of the languages presented so far, it is easily understandable, even for novices. However, in the example, the keywords use more characters than the actual data. This can make that data difficult to spot between the keywords.

```
concept Server {
    mandatory feature FileSystem {
        someOf { feature NTFS; feature APFS; feature EXT4;}
    }
    mandatory feature OperatingSystem {
        oneOf {feature Windows; feature macOS; feature Debian;}
    }
    feature Logging;
    constraint Windows -> NTFS;
    constraint macOS -> APFS;
}
```

Listing 3.9: Server example expressed in VELVET.

VM In an industrial project, a variability modeling approach called VM [AAG+19] has been created. It is specifically tailored to the video domain, but it can also be used for more general variability modeling. In Listing 3.10, we show our running example expressed in this language. It also uses nesting with curly braces and aims at limiting the number of keywords. As most languages presented thus far, it stores the constraints in a separate list at the bottom. However, it only marks the beginning of a new section with a keyword. This is a way to save keywords, especially when there are many constraints in a file, but still maintaining readability.

```
Relationships:
Server {
    FileSystem { someOf {
        NTFS APFS EXT4
    }}
    OperatingSystem { oneOf {
        Windows macOS Debian
    }}
    ? Logging
}
Constraints:
Windows requires NTFS
macOS requires APFS
```

Listing 3.10: Server example expressed in VM.

https://doi.org/10.24355/dbbs.084-202008211051-0

IVML The INDENICA Variability Modeling Language (IVML) [SKE18] aims at being more similar to programming languages, instead of merely representing feature models. It uses fields, assignments, and enumerations to describe both the variability and the configuration. Thus, the example is implemented in a very different way, see Listing 3.11. The leaf features are realized using the values of enumerations. These values could then be assigned to the corresponding fields when creating concrete configurations. Still, the constraints restrict what can be assigned to those fields. Though this language is more expressive than most others, it is also the most unintuitive to represent plain feature models in. The tree structure from the feature model is lost. Instead, there are typed variables for features. Additionally, the constraints over possible variable assignments require the most characters of all presented examples.

```
1  project Server {
2      compound Server {
3          enum FileSystem {NTFS, APFS, EXT4};
4          enum OperatingSystem {Windows, macOS, Debian};
5
6          Boolean logging;
7
8          FileSystem fileSystem;
9          OperatingSystem operatingSystem;
10         operatingSystem == Windows implies fileSystem == NTFS;
11         operatingSystem == macOS implies fileSystem == APFS;
12     }
13 }
```

Listing 3.11: Server example expressed in the INDENICA Variability Modeling Language (IVML).

3.3.2. Comparison of the Presented Languages

After introducing various textual variability languages, we now summarize and compare the different design decisions of the presented languages. For this, we consider mostly the design of the concrete textual syntax, as a comparison of their different language features has already been done in ter Beek et al. [tBSE19].

We show an overview of the languages and their design choices in Table 3.2. One can see that three languages use references to represent the hierarchy, while nine use nesting. Those languages that use references, do not have additional blocks to mark. Instead, all the information regarding a feature is present in a single line. The other languages use either curly braces, indentation, parentheses, or square brackets to mark the extent of blocks. Curly braces are most common with five languages using them. Indentation is used in two languages. Parentheses and square brackets are used by only one language each.

Three languages only use symbols or single character keywords to be as short as possible. The other languages use full-length words as keywords, but vary in how many keywords per language construct are used. VELVET might be the wordiest language, which even uses

https://doi.org/10.24355/dbbs.084-202008211051-0

Language	Hierarchy	Blocks	Keywords	Line Endings	Constraint Location	Location of Groups
FDL	reference	none	full	new line	separate	parent
GUIDSL	reference	none	symbols	semicolon	separate	parent[1]
FAMILIAR	reference	none	symbols	semicolon	separate	parent
PyFML	nesting	[]	full	semicolon	separate	parent
Clafer	nesting	indentation	minimal	new line	in-line	parent
SXFM	nesting	indentation	symbols	new line	separate	between
VSL	nesting	()	minimal	semicolon	separate	between
TVL	nesting	{}	full	semicolon	in-line	between
μTVL	nesting	{}	full	semicolon	in-line	between
VELVET	nesting	{}	full	semicolon	in-line	between
VM	nesting	{}	minimal	new line	separate	between
IVML	nesting	{}	full	semicolon	in-line	n/a

[1] for *or* groups also the parent's parent

Table 3.2.: Summary of design decisions of the presented languages regarding the concrete syntax.

the keyword 'feature' for each feature. Most other languages minimize the use of keywords by deploying sensible defaults or combining several similar elements under one keyword. For instance, by having a section for constraints instead of writing 'constraint' before each constraint.

Semicolons are used by eight languages to end lines. Only four languages do not use line delimiters. In contrast, the distinction between whether to store constraints in a separate list or as children beneath features is distributed evenly over the languages. Seven languages have a separate list, while five store them as children.

Five languages specify the location of groups or cardinalities at the parent, while six store this information between the parent and the children. For this, we also count the GUIDSL grammars as part of the first category, where the information about *or* groups is distributed between both the parent and the parent's parent. IVML is excluded here, as it does not have a concept of groups.

3.4. Questionnaire From the MODEVAR 2020 Workshop

We created a questionnaire that has been conducted at the MODEVAR 2020 Workshop in Magdeburg, Germany. This way we could gather the assessments, preferences, and needs of the individual members of the SPLE community beyond what was covered in the usage scenarios and the existing languages. The participants worked in pairs on the questionnaire, so they could discuss the questions, their preferences, and give well-thought-out answers. We give an

https://doi.org/10.24355/dbbs.084-202008211051-0

overview of the questions we developed in Section 3.4.1 and discuss the results in Section 3.4.2.

3.4.1. Overview of the Questionnaire

The survey starts with questions concerning the most accessible aspect of the language — its concrete syntax. The concrete syntax is what the user has to read and write in the end. Furthermore, people tend to have strong preferences on certain aspects of the concrete syntax. Thus, when presented with a concrete example, participants can directly grasp what the question is about and can give their opinion. This is in contrast to thinking about more abstract topics, where much more imagination and indirection is required from the participant to obtain an answer. We choose to start with examples of different decisions for the concrete syntax to ease into the more abstract questions. The following sections give an overview of the questions in the order of appearance in the questionnaire. For reference, we also include the complete list of questions as they appeared in the questionnaire in the appendix (cf. Appendix A.1).

Since we expected the total number of responses to be small, we wanted to maximize the qualitative output in addition to the quantitative results. Hence, for each question, we ask for the reasoning behind the given answers. This includes providing reasons for their answer as well as reasons against the other answers. Also, we provide an additional text field for comments for each question and the questionnaire as a whole. Finally, we give participants the option to provide their names, so we can quote their comments here.

Q1: Keyword Length

Keywords can be long, naming the thing they represent, very short symbols, or single characters. The former is easy to understand when read, but might be cumbersome to write without adequate editor support. Also, it adds visual noise and can make lines unnecessarily long compared to the actual content the user represents. This also increases the file size, although nowadays space is no longer a huge concern, especially with the ability to compress the files. Using a symbol as a keyword is very short and only adds one character to the length of a line or the size of a file, but usually does not convey any meaning by itself. A user has to memorize the meaning of every symbol to quickly edit a file. This introduces a steep learning curve that is off-putting for novices. It might still be beneficial for experts, as it enables very fast reading and writing once accustomed. A third option is a compromise between long and short keywords. Abbreviated keywords (e.g., “alt” instead of “alternative”) might convey enough meaning for novices to be able to quickly learn the language, but still be short enough so experts will want to write them. Which of these options is best for the language is strongly dependent on the envisioned use cases and the preferences of the users.

Q2: Line Breaks

In many languages, such as Java or C, lines are terminated with a semicolon. Other languages, such as Python or Bash, do not require such a character. For very long lines that should be split into multiple lines, this means that an escape character for the line breaks is required. None of

https://doi.org/10.24355/dbbs.084-202008211051-0

those solutions are ideal, as they force the user to either explicitly state their intent to end a line or to continue a line. We ask which of these approaches the participants prefer.

Q3: Structuring

When expressing information that logically belongs to another element (e.g., attributes or children of features), the beginning and the end of this 'block' has to be marked to set it apart from other elements. This is assuming it is not done by referencing the element it belongs to. There are different ways to mark these blocks. It can be realized with indentation as in Python or using special characters to mark the beginning or the end. Java uses curly braces, while LISP uses parentheses. We ask which of these options should be used.

Q4: Hierarchy

Feature models are inherently hierarchical, so this has to be reflected in the language. To represent hierarchies in a textual format, one can use nested blocks to represent children or use references to represent the hierarchy as a graph (cf. Section 3.3). Both forms have their advantages and disadvantages. For small models, combining nesting with proper indentation is an intuitive way to represent hierarchies, as it can be comprehended at a glance. However, this approach does not scale very well. With bigger models, nesting results in a large amount of indentation for the deeper levels. The added whitespace displaces the actual information. To this end, files become large and hard to read. Also, when many children appear between two sibling features in the file, the information that these elements are indeed siblings, cannot easily be seen anymore. However, this concern is mitigated by editors that support the folding of sections with more indentation. References from parents to children scale better in this regard but add their own disadvantages. There is added redundancy through the repeated names in the references, which also introduces a possible source of errors when editing files manually. The tree structure cannot easily be seen. Instead, it has to be reconstructed by tracing through the references. We ask if the hierarchy should rather be represented using nested blocks or as references to children.

Q5: Group Membership

All the different languages we introduce in Section 3.3 have support for groups or cardinalities, so the question arises, where to put the corresponding keyword. In principle, there are three possibilities: they can be located at the parent, at the children, or in between. At the parent (the semantic being 'I have an *or* group') is the least flexible, as there is no way to specify on a per-child basis whether they should be mandatory or optional. More flexible is the specification at the children, with a semantic of 'I belong to an *or* group', as it allows one to have different group types beneath the same parent feature. However, this introduces some redundancy and a possible source for errors, as the same group type has to be repeated for every child. The most flexible solution is to have the keyword in between. This way, one can even have multiple groups of the same type beneath one parent and eliminate the redundancy of the previous approach.

https://doi.org/10.24355/dbbs.084-202008211051-0

However, depending on whether nesting or referencing is used, this variant introduces another level of nesting or indirection, so none of the solutions is ideal. Feature diagrams are actually inconsistent here, as the marker for mandatory or optional is on the child, while the markers for *alternative* and *or* groups are at the parent. If the goal was to be as close to the notation of feature diagrams as possible, one could adopt this same inconsistency.

When making decisions about the concrete syntax, this often has implications on the abstract syntax, too. However, the following questions are explicitly more abstract, concerning the scope, expressiveness, and decisions regarding the realization of the language. Although the overall structure of the abstract syntax can be reused from the de facto standard notation of feature diagrams, there are still many details to be considered.

Q6: Groups, Cardinality, and Constraints

The next question concerns the expressive power of groups. More specifically, whether groups are expressive enough or if something more general, such as group cardinalities or constraints (e.g., at least or at most n elements out of a set of features), is needed. The groups *or* and *alternative* are special cases of cardinalities [CHE05] and are both simpler and more limited in their expressiveness than cardinalities. A restriction to groups might be desirable to aid learnability and analyzability, but might also be too restrictive in some cases.

Q7: Scope

This question aims at determining, which language features are necessary, nice to have, or just not needed and too complex. For each of the following items, participants can choose on a scale between 'absolutely necessary' and 'strongly against' plus 'not familiar with the concept', in case participants are only familiar with some of the concepts:

- **Default selections.** Features could be marked as default, so they are preselected when creating a new configuration in a configuration editor (cf. Section 3.2).
- **Abstract features.** Features that do not have a mapping to an implementation artifact, but are intended solely for structuring, could be marked as abstract features in contrast to regular concrete features (cf. Section 2.1.3).
- **Save entire configurations.** In addition to the feature model describing the variability, entire configurations describing instances in the space of variability could be persisted in the syntax.
- **Attributes for features.** Selections of features are boolean in nature. Some languages, however, allow for additional attributes with different data types, such as integers.
- **References to other feature models.** As a simple mechanism for composition, another feature model could be referenced and included as a subtree (cf. Section 3.2).
- **Feature model interfaces.** A more sophisticated mechanism for composition, aiming at enabling better scalability, is to use interfaces, similar to interfaces in object-oriented programming [SKT+16].

https://doi.org/10.24355/dbbs.084-202008211051-0

- **Extension mechanism for arbitrary data.** Tool-specific data (e.g., layout information) or other details in a given notation could be added to a model using a generic extension mechanism, instead of trying to provide a specific syntax for each possible datum.

Q8: Expressiveness of Constraints

This question is about how much expressive power is required for constraints. Participants decide whether propositional logic is expressive enough for their purposes or if first-order logic or something even more expressive is needed. Limiting the language to propositional logic would be a benefit, as it is the simplest to analyze.

Q9: Separation of Concerns

For each piece of information, we have to decide where it should be stored in the file. This could be in-line, where the feature itself is stored (i.e. as a keyword or in a nested block), or separately. Separately could be a separate list or section in the same file or a separate file entirely. When the information is stored separately, the link to the corresponding feature or features has to be established by reference. This reference can point from the feature to the information or from the information to the corresponding feature. For instance, constraints are often listed separately in the same file and point to the features they use. In languages that use references to represent their hierarchy, the parent feature has pointers to the children, demonstrating the inverse direction of references. With this question, we determine, how the community would categorize different kinds of information from the abstract syntax. It would be interesting to find a consistent rule behind the categorization, but there might not be any. These are the different kinds of information we ask participants to categorize:

- **Hierarchy.** How the hierarchical tree structure should be represented.
- **Abstract feature flag.** Whether the feature has a corresponding implementation (concrete) or is only for structuring (abstract).
- **Constraints.** Cross-tree constraints that have to be fulfilled in addition to group membership or cardinalities.
- **Default selections.** Preselected features in a configuration editor.
- **Configurations.** The specification of which features are included in a specific product.
- **Arbitrary additional data.** Tool-specific data (see above).

Q10: Reuse of Existing Formats

To minimize the implementation and integration effort, one could use or adapt an existing serialization format. Examples for these include XML, YAML, JSON, EDN, and OpenDDL. However, using an off-the-shelf format has considerable drawbacks. As they are made for generic data and lack knowledge of the domain, they tend to be more verbose than a DSL. Also, they have no support for specialized constructs such as constraints, so they would either have

https://doi.org/10.24355/dbbs.084-202008211051-0

to be written as a string or as the AST of the constraint. Neither option is ideal. Still, it is interesting what the community thinks about reusing existing formats.

Q11: Tool Lock-In

The final question is about the trade-off between IDE support and tool lock-in. We ask the participants how much IDE support they want and, in return, how much tool lock-in they are willing to endure. The spectrum of possible answers ranges from no tool support and only an EBNF specification to a projectional editor with maximal tool lock-in.

3.4.2. Results of the Questionnaire

There are ten submissions to the questionnaire. As mentioned before, the participants worked in pairs on the questions, so in total 20 participants worked on the questionnaire. We go over the questions again, presenting the results as well as commenting on interesting observations.

Q1: Keyword Length

As shown in Figure 3.1, five out of ten answers voted for long keywords. Some comments mentioned that symbols were confusing, readability was a high priority and long keywords were okay to type with IDE support. Two participants voted for abbreviated keywords and symbols, respectively. Abbreviated keywords have been called a perfect mix. One participant was undecided, saying that it depends on the purpose of the language, but either long keywords or symbols should be used. This result is consistent with the analyzed existing languages, where most languages also use long keyword names.

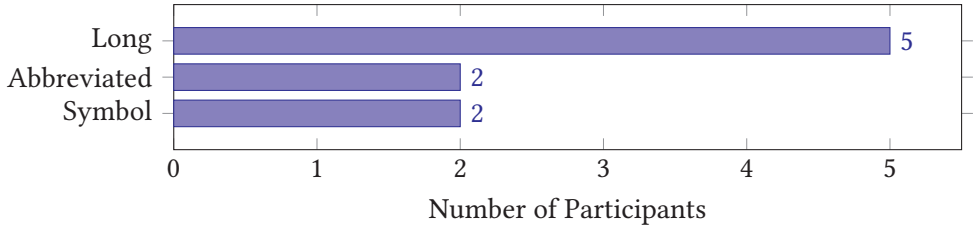


Figure 3.1.: Voting results on the length of keywords.

Q2: Line Breaks

Whether to end lines with a semicolon or to use line breaks resulted in a tie. Five voted for semicolons, while the other five submissions voted for line breaks, as depicted in Figure 3.2. Line breaks were said to look cleaner, lines might not get too long and semicolons were meant for programmers, not for product-line engineers. On the other hand, Gilles Perrouin and Michael Nieke claim that the backslash that is commonly used to break lines is misleading and potentially very unintuitive for non-experts. Conversely, most of the existing variability languages use semicolons to break lines.

https://doi.org/10.24355/dbbs.084-202008211051-0

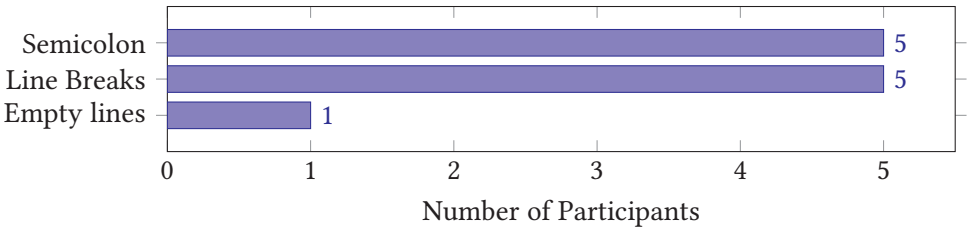


Figure 3.2.: Voting results on whether to end lines with a semicolon or break long lines.

Q3: Structuring

Regarding the structure, indentation was more popular than curly braces with six votes versus four, see Figure 3.3. The concept of indentation to structure files was said to be well-known to most potential users. One participant suggested that curly braces could be allowed, but not made mandatory. This, however, would violate one of the general design guidelines to avoid redundant concepts. Also, from a parser's view, this would not solve any issues with ambiguities, as it must still be possible to parse both concepts unambiguously. Two participants voted for parenthesis. The LISP-style was called trivial to parse and easy to edit with the appropriate editor mode. This result is in stark contrast with the existing languages, where only two out of the twelve languages rely on indentation for structuring.

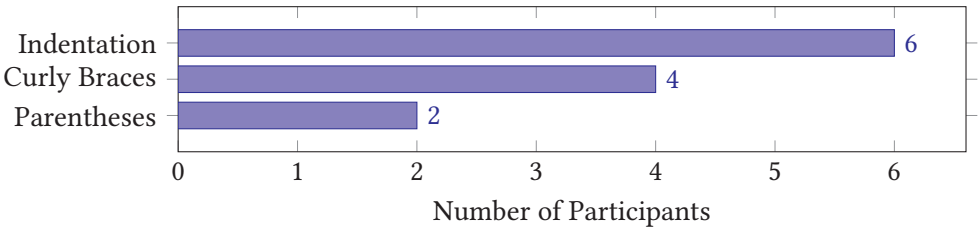


Figure 3.3.: Voting results on which means of structuring to use.

Q4: Hierarchy

Nesting versus references was a close match with five submissions in favor of references and four for nesting. Two submissions pleaded that both should be allowed. Arguments were voiced in favor of references, claiming better scalability and no need for special handling of references into other models. As a counterargument, the added redundancy when using references was mentioned. Again, compared to the existing languages, this result is surprising, as there are only three out of the twelve that use references. Figure 3.4 visualizes the results.

Q5: Group Membership

Figure 3.5 shows that it was quite clear to the community, where to put groups. Only one submission voted for 'at the parent', one suggested in the constraints, two said on the children,

https://doi.org/10.24355/dbbs.084-202008211051-0

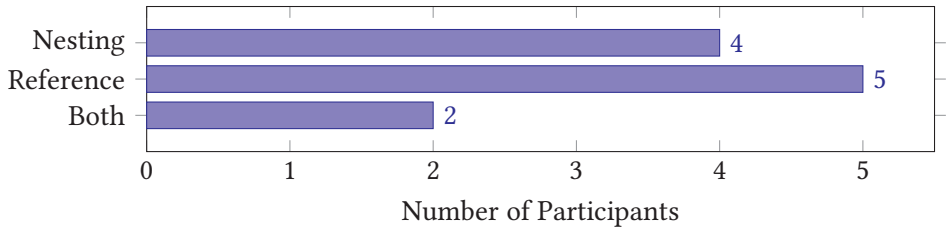


Figure 3.4.: Voting results on how to represent hierarchy.

whereas seven voted for 'in-between'. With one submission calling for the possibility to have the group keyword on the same line as the first item when using in-between groups.

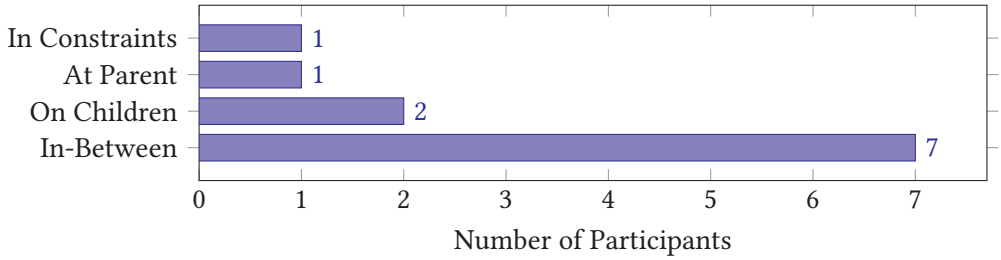


Figure 3.5.: Voting results on where to locate the group keyword.

Q6: Groups, Cardinality, and Constraints

Many submissions opted for multiple mechanisms to specify the group type of child features, as shown in Figure 3.6. Thus, there are six votes for simple groups, seven votes for cardinalities and four votes for constraints on at least or at most n features out of a specific set. One comment by Gilles Perrouin and Michael Nieke indicated that the proposed constraints, though interesting, might be hard to reason about, so they would belong into a higher language level.

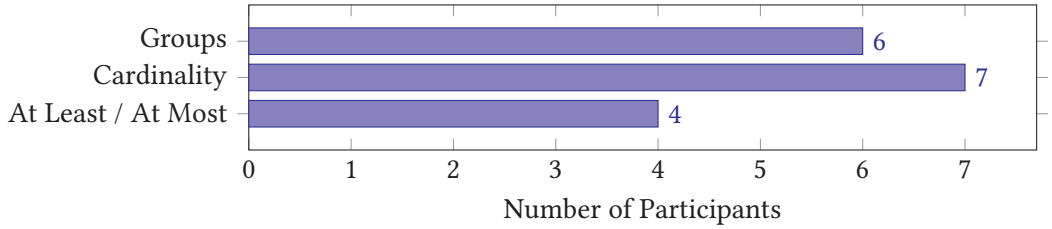


Figure 3.6.: Voting results on whether groups are expressive enough or if cardinalities or specialized constraints are required.

Q7: Scope

In this question, the participants voted whether presented language features should be included in the language or not. Figure 3.7 shows the detailed voting results. The community tends

https://doi.org/10.24355/dbbs.084-202008211051-0

slightly towards including default selections for configurations. Abstract features are wanted by the community, although one comment noted that it might simply be a special case of more general attributes for features. For the ability to save configurations, participants deem it beneficial to specify a default way, but not in the main model or the main language. The community favors attributes for features. References to other models should also be included. Interfaces, in contrast, were deemed too complex for the language or the concept was not understood. This is in line with the state of research on interfaces for feature models, which is still in a very early stage, where much is yet to be understood and defined [SKT+16]. Also, most submissions voted in favor of an extension mechanism for arbitrary additional (tool-specific) data.

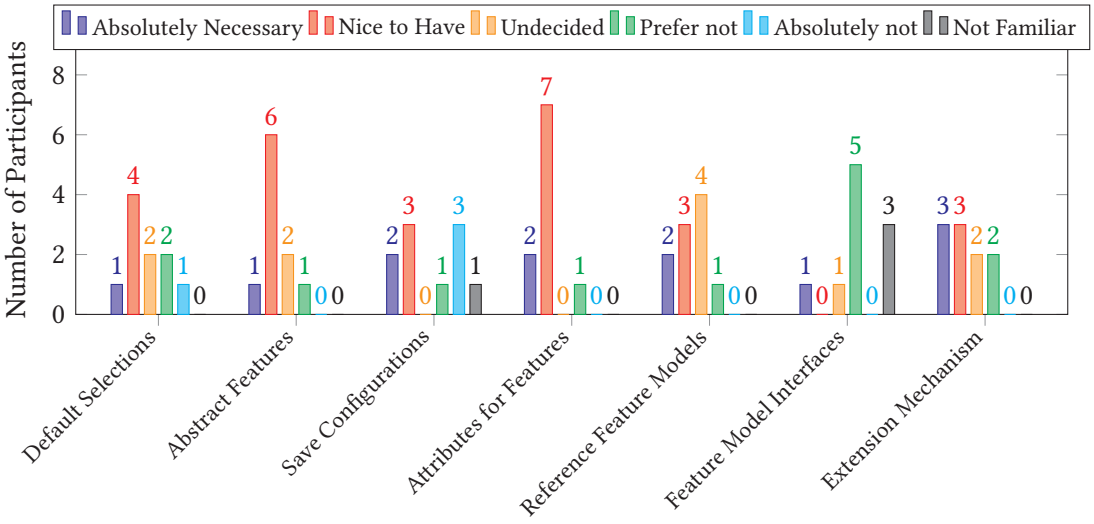


Figure 3.7.: Voting results on which language features should be part of the scope of the language.

Q8: Expressiveness of Constraints

Propositional logic should be included, as nine out of ten submissions voted in favor of it (Figure 3.8). This is in line with the findings by Knüppel et al. [KTM+17], showing evidence that simple require and exclude constraints are not expressive enough for real-world models and that propositional logic is indeed required. Three submissions also voted for first-order logic. However, the comments say that it is not meant for the basic language level, but should be available with limited functions in higher levels to be compatible with existing languages.

Q9: Separation of Concerns

For this question, there were different kinds of information, where the community should vote on whether to specify them in-line, by reference in the same file, or in separate files. We show the whole range of answers in Figure 3.9. In summary: according to the submissions, both the hierarchy and whether features are abstract should be specified in-line. Constraints were almost unanimously voted to be put in a separate list within the same file. This is consistent

https://doi.org/10.24355/dbbs.084-202008211051-0

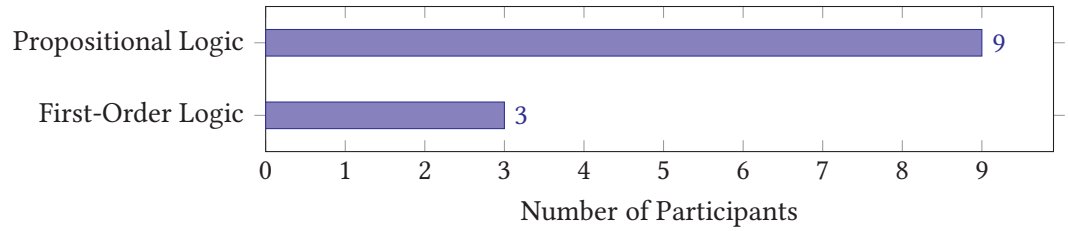


Figure 3.8.: Voting results on the required expressiveness of constraints.

with existing languages, most of which have a separate list of constraints. The community was undecided on the best location of both default selections and additional data. However, entire configurations should be stored in separate files. One comment suggested that a criterion for what should be specified at the feature could be whether the information is regarding one or multiple features. This would explain why the product-line researchers assign abstract features to be specified in-line but place constraints in a separate list.

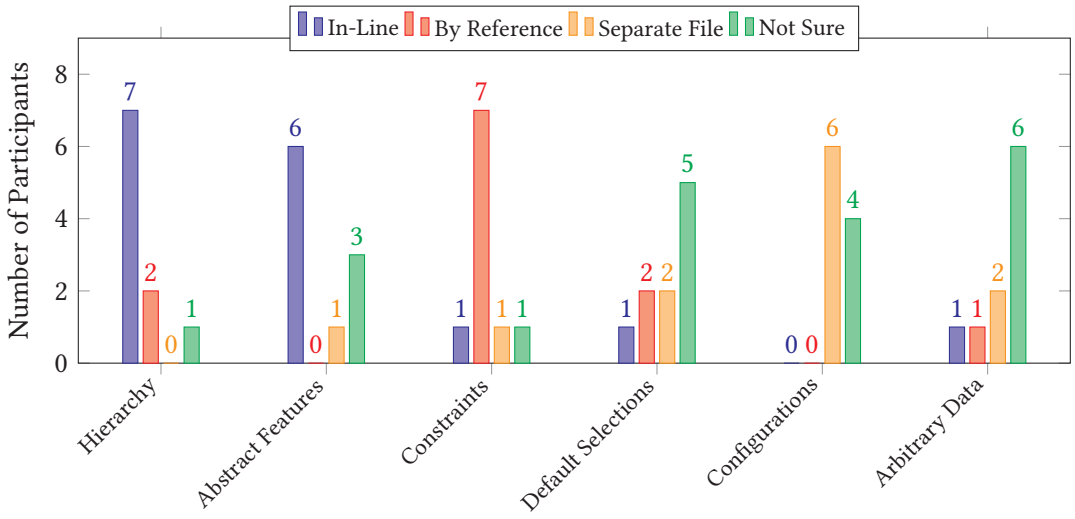


Figure 3.9.: Voting results on what to specify in-line, by reference or in separate files.

Q10: Reusing Existing Formats?

Whether to use an existing format got mixed responses (Figure 3.10). Four submissions were undecided with one of them tending more towards a no. Another four submissions voted for yes, although they were split on the format to use. JSON, YAML, XML, and a compromise, such as SXFM with a DSL inside an XML tag, were suggested. Two submissions voted against using an existing format, because “we already have that”, probably referring to SXFM or tool-specific formats, such as FeatureIDE’s XML.

https://doi.org/10.24355/dbbs.084-202008211051-0

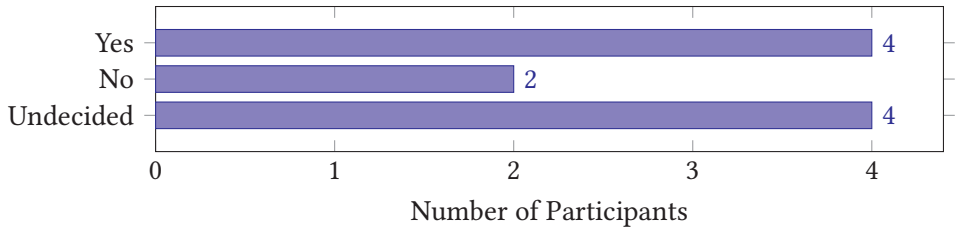


Figure 3.10.: Voting results on whether to reuse an existing serialization format.

Q11: Tool Lock-In

Lastly, regarding tool support, no one voted for projectional editors. A compromise between no and full editor support was the most popular option. Six submissions voted in favor of only an EBNF specification, seven found a small independent default library valuable, while four submissions went for the full IDE support and tool lock-in. Figure 3.11 shows the voting results.

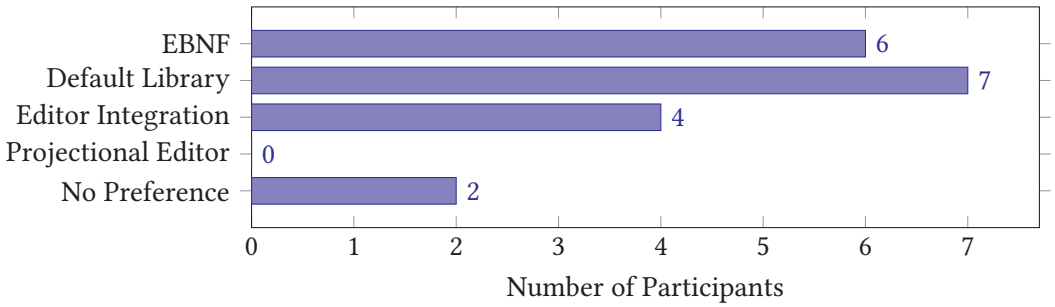


Figure 3.11.: Voting results on the level of tool lock-in that is tolerable.

One general observation we made when evaluating the survey was that some participants seem to envision the language to be similar to a programming language, such as Python or Java. They seem to choose whichever they are most familiar with. In contrast, others seem to have different stakeholders in mind and deem concepts from programming languages too technical.

3.5. Summary of the Requirements

In this chapter, we looked at general design guidelines for DSLs, the collected requirements from the community [BC19], existing textual variability languages, and the results from our questionnaire. Now we summarize the observations and recommendations for our language.

From the 14 collected requirements, we can conclude that exchange, mapping to implementation, teaching and learning, and storage are the most important scenarios to the community with actionable requirements. It follows that the language should be simple so it can be easily integrated into various tools and taught at universities. At the same time, it should be expressive enough to represent real-world models that can be used for benchmarking and everyday work of product-line engineers. Though the importance of decomposition and composition is disputed

https://doi.org/10.24355/dbbs.084-202008211051-0

among the community, we argue that it is an important requirement for scalability to large, real-world models. Thus, we also consider it in the language design.

The analysis of twelve existing languages showed that there is a vast number of individual design choices in these languages. Still, every language combines them in a slightly different way. These decisions sometimes seem to be based on preference, while others are born from a specific need for a project. All of the languages have been introduced to the community with a paper, explaining the reasoning behind the language, but not resulting in widespread acceptance. By collecting requirements, discussing, and gathering feedback from the whole community, acceptance might improve. From our questionnaire at the MODEVAR 2020 workshop, we can deduce that there are some decisions that product-line researchers seem to agree on, but on others, they are completely divided. Combining and weighing these sources of opinions, we arrive at the following design decisions that guide the definition for our language.

Regarding the concrete syntax of the language:

- Full-length keywords are preferred over short symbols.
- Indentation is preferred over curly braces.
- Both the usual group keywords as well as cardinalities should be available.
- It is unclear, whether to use references or nesting to represent the hierarchy.

Regarding the scope:

- The language should support abstract features.
- Default selections for configuration processes should be available.
- Attributes should be present, although without a type system and not within constraints, to keep the language analyzable with a SAT solver.
- There should be an extension mechanism for tool-specific data.
- Simple references need to be supported to enable the decomposition of large models.
- Propositional logic should be the expressiveness of constraints.

Regarding the implementation: To enable integration into other tools, both an EBNF specification and a small default library with a parser and a printer should be available. Integration into editors with syntax highlighting would be nice, but should not add to the size or the dependencies of the default library.

With these requirements set and an insight into the needs and preferences of the community, we propose specific concepts for a language in Chapter 4, closely adhering to the requirements. We continue by implementing tool support for the language in Chapter 5, before evaluating the approach in Chapter 6.

https://doi.org/10.24355/dbbs.084-202008211051-0

4. Proposal for a Universal Variability Language

After examining, weighting, and condensing different sources of requirements, we now show our proposal for a language. The community was split on whether to use nesting or references to represent the hierarchy of feature models, with valid and important arguments for both sides. Thus, we present both approaches in two different language concepts. This way, we can compare them side by side and see which the community prefers in the evaluation (Chapter 6).

First, we present the characteristics that are common to both language concepts in Section 4.1. Then, we show the concept that uses nesting to represent the hierarchy in Section 4.2. In Section 4.3, we show the other concept that uses references. As both concepts share the same means of decomposition and constraints, we introduce those afterward in Section 4.4 and Section 4.5, respectively.

4.1. General Characteristics

Features and their hierarchy are the basic elements in both feature models and all of the presented languages in Section 3.3. Thereby, they should play a central role in our language as well. This is in accordance with the guideline to reuse concepts from other languages whenever viable (cf. Section 3.1). For readability, we use full-length keywords, but avoid having to write too many of them, when they can be omitted. This reduces the effort of writing out those keywords and shrinks the file size for storage. For instance, we do not include a `feature` keyword that has to be repeated for every feature. Instead, we opt for the single keyword `features` that marks the beginning of the feature section in the file. In that section, unless some keyword or context dictates otherwise, every new line will denote a new feature per default.

The names of features start with a letter, optionally followed by other letters, digits, or underscores. Additionally, we assume feature names to be unique within a file, so referencing by name can be done without ambiguities. We avoid allowing exotic characters, such as whitespace or other special characters. This way, identifiers in our models can safely be converted into most other formats, although not vice versa. Translating names with exotic characters to our language can be achieved by creating attributes (e.g., a description or a display name, see below). The only format of the analyzed existing languages that has more restrictive identifiers is FDL. It requires identifiers of leaf features to start with lowercase letters and other feature names to start with uppercase letters (cf. Section 3.3.1). Thus, translating identifiers into FDL would

https://doi.org/10.24355/dbbs.084-202008211051-0

require adjusting the case.

Cardinalities are allowed in addition to the usual groups *or* and *alternative*. Cardinalities have been very popular in the first questionnaire (cf. Q6 in Section 3.4.2) and allow more flexibility than groups without introducing much complexity. For the concrete syntax, we reuse the notation from the UML specification [Obj17, Sec. 7.5.4.1]. That is `[1..*]` for the *or* group and `[1..1]` for the *alternative* group. When lower and upper bounds are equal, as it is the case for the *alternative* group, the shorthand `[1]` can be used.

Since most participants from the first questionnaire found abstract and default features as well as attributes and an extension mechanism for arbitrary data useful, we include these features into the language. However, to avoid complexity we combine all those into the single concept of attributes. Attributes in our language are, in essence, associative arrays, also known as maps or dictionaries in programming languages. They can be attached to any feature and contain pairs of keys and values. Keys are always identifiers. Values can have typical primitive data types, present in most programming languages. These consist of booleans, strings, integers, and floating-point numbers. For simple aggregate data types, vectors and maps are available. By convention, the attributes "abstract" and "default" can denote abstract and default features, respectively. In turn, we expect boolean attributes to be used often, so for boolean attributes with the value "true", one can omit explicitly stating the value to make the code more concise. Other use cases for these attributes include the specification of constraints, mapping information from a feature to its implementation, more detailed descriptions of a feature, or any other tool-specific data. This way, users can choose whether they want to list all the constraints separately at the end of the file or beneath corresponding features. For now, attributes can only have static values and can not be used in constraints to keep the language simple to analyze using SAT solvers, as more powerful SMT solvers are often much slower for certain problems [TSS19]. A type system could be added later for a second language level. With it, one could specify types (e.g., integers) for specific variables in attributes, which could then be used in constraints and initialized during configuration. This would only require little new concrete syntax to learn but call for an SMT solver to still analyze the model.

Specifying whole configurations is out of scope for this language concept, as the configuration process is a second, distinct step from modeling the variability. Also, participants of the questionnaire agree that it belongs into a separate artifact. Thus, we do not include syntax for it. In the first questionnaire, the community was split on whether to include it or not. When one language for describing variability gains acceptance, a separate language for describing entire configurations that matches the style of the first one could still be added. We also do not include a concept of interfaces between models (cf. [SKT+16]). This concept was deemed too complex by the community and the research is only in its early stages. However, we include a concept for composition and decomposition using references. We describe this concept in Section 4.4.

Constraints support the expressive power of propositional logic. This is simple enough to be analyzable with SAT solvers, but complex enough to represent most real-world cases [KTM+17]. Also, this is what has been voted for almost unanimously in the first questionnaire. We show the syntax of constraints in Section 4.5.

https://doi.org/10.24355/dbbs.084-202008211051-0

4.2. Language Concept 1: Nested Hierarchy

In addition to the previously described general characteristics, the following design decisions apply to the first language concept: We use nesting and indentation, so we refer to it as the nesting language concept. Indentation was the most popular option to mark the extent of blocks in the first questionnaire and was said to be well-known by every possible audience. New lines do not have to be marked specially, as no semicolons are required. The voting results in the first questionnaire on this point were split half and half (cf. Figure 3.2), so they do not provide a preference. Semicolons are used in the C-like family of languages, but with nesting and indentation this language is more similar to Python, so we adhere to its style and omit semicolons. This is per the design guideline to reuse existing notations as much as possible, to not overwhelm experts with new details to learn for the new language (see Section 3.1).

Attributes can be specified in curly braces and can belong to any feature. Within the curly braces, pairs of a key and a value can be specified. Values have a syntax that is similar to most programming languages or textual exchange formats such as JSON.

The location of groups is between the parent and its children. This option is the most flexible and introduces the least amount of redundancy. It is also the most prominent choice of the existing textual languages, especially when considering those that use nesting, and was voted for most often. We also locate the keywords for mandatory and optional features between the parent and the children, as opposed to at the children themselves, as it is common in feature models. This leads to a more consistent notation and reduces redundancy and file size, as those keywords have to be used only once per group. Thus, the available group types in this language concept are *optional*, *mandatory*, *or*, *alternative*, and cardinalities, as described in the previous section. One of these options has to be specified for each set of child features, as there is no implicit default *all* group as in most other languages. A default group to use, without thinking about the final cardinality, is the *optional* group. This design decision simplifies later refinement, as the group type could be changed without modifying the children's level of indentation. Also, children can be moved between groups more easily when all groups use the same nesting level.

Listing 4.1 shows the server example from Figure 2.2 represented in the nesting language concept. The file starts with the `features` keyword. Beneath that, all the features from the example are specified. The abstract features have the boolean attribute `abstract` in curly braces, omitting the value `true` for brevity. To show other possible uses with more attributes, we add two attributes to the logging feature that are not present in the feature diagram from Figure 2.2. The attribute `default` indicates that configuration editors should select this feature per default when starting a new configuration. The attribute `log_level` demonstrates the use of a string as the data type. In a separate section, the constraints for the model are listed with one constraint per line. These constraints could also have been added beneath other features as attributes. Even though this way, we introduce two locations to specify constraints, we think that both are justified. For small models, a short list of constraints at the end of the file can be a good overview. For large models with many constraints, however, this list would become unmanageable. In that case, we argue that placing constraints near their referenced features

https://doi.org/10.24355/dbbs.084-202008211051-0

```
features
  Server {abstract}
    mandatory
      FileSystem
        or
          NTFS
          APFS
          EXT4
      OperatingSystem {abstract}
        alternative
          Windows
          macOS
          Debian
    optional
      Logging {
        default,
        log_level "warn"
        // Attributes is the extension mechanism
        // for arbitrary data.
      }

constraints
  Windows => NTFS
  macOS => APFS
```

Listing 4.1: Server example represented in the nesting language concept.

can be a better alternative. Different rules of placement could be applied, depending on the project's needs. One option is to place a constraint beneath one of the features that it references. Another option is to place it beneath the feature that is deepest in the tree but still has all the constraint's referenced features as part of its subtree.

4.3. Language Concept 2: Referenced Hierarchy

The main difference to the previous concept is that we now use references for the hierarchy, instead of nesting and indentation. We refer to this second language concept as the referencing language concept. The concepts regarding attributes still apply here, although they have a different syntax. Curly braces are no longer required to distinguish attributes from features. Instead, attributes can be added with indentation in separate lines. This is similar to the way child features are specified in the nesting language concept. This is made possible, because child features do not need to be specified in an indented block, but are referenced in the same line as the referencing feature. Thus, the indented block can be used for attributes instead. In case there are only one or two short attributes, specifying them in separate lines could seem superfluous. Thus, few attributes can be appended after the referenced children, separated by commas, resulting in a more compact appearance.

https://doi.org/10.24355/dbbs.084-202008211051-0

Line breaks are the same as in the previous language concept. They do not have to be marked explicitly with a semicolon. Instead, each feature specification is expected to start on a new line.

Listing 4.2 shows the code snippet. Again, the file contains two sections labeled `features` and `constraints`. The root feature defines three children, with an *all* group. This is the default group when no special keyword or cardinality is given, the same as in feature diagrams. Mandatory features are marked with an exclamation mark. Since it has only one attribute (`abstract`), it is appended in the same line after a comma. Each line refines previously defined features, adds children, and attributes. For the logging feature, we add the same attributes as before to demonstrate the syntax.

```
features
Server: FileSystem! OperatingSystem! Logging, abstract
FileSystem: or NTFS APFS EXT4
OperatingSystem: alternative Windows macOS Debian, abstract
Logging
  default
  log_level "warn"

constraints
Windows => NTFS
macOS => APFS
```

Listing 4.2: Server example represented in the referencing language concept.

4.4. Composition Mechanism

Most participants of the questionnaire saw the necessity to split up feature models to cope with ever-growing real-world models (cf. Figure 3.7). That is why we include the following system for decomposition. The syntax and mechanism for decomposition and referencing is the same for both language concepts, so we show it only once. It is meant to be simple and familiar, as it is very similar to Java's package and import system.

We move the file system and the operating system from the example model into separate files with their own namespaces, so they can be imported and reused from the server or other feature models in the future. Listing 4.3 shows the corresponding code snippet for the file system. At the top of the file is a namespace specification, which is equivalent to the Java package declaration. The namespace identifier can then be used in other files to import it. For convenience and simplicity in small examples, the namespace specification can be omitted when it would be the same as the root feature name. We show this in Listing 4.4.

Imports reference the namespaces, so features from those models can be referenced by writing `namespace.Feature`. To avoid having to write fully qualified names whenever an imported feature is used, namespaces can be aliased with the `as` keyword. Also, with the `refer` keyword, specific or all features can be referred, so they can be used without any namespace prefix later.

https://doi.org/10.24355/dbbs.084-202008211051-0

```
namespace org.featuremodels.FileSystem

features
  FileSystem
    or
      NTFS
      APFS
      EXT4 {default}
```

Listing 4.3: Example decomposition of the file system from the server example in the nesting language concept.

```
// namespace OperatingSystem can be omitted here
features
  OperatingSystem {abstract}
    alternative
      Windows
      macOS
      Debian {default}
```

Listing 4.4: Example decomposition of the operating system from the server example in the nesting language concept.

We show that syntax in Listing 4.5, where we compose the previous two submodels into the entire server example.

Semantically, referencing a feature from another namespace can be understood as inserting it and its subtree at the place where it is referenced. These semantics are not important for the syntax itself, but for understanding models and converting or importing them into other languages or tools that might not support the composition of multiple models. When including a subtree, any constraints that are involved with something from that subtree (even transitively) will have to be considered, too.

With this syntax, the question arises whether it should be allowed to include only a subtree of another model when importing it or if the whole submodel including its root feature must be referenced. While the former would be more flexible and could even allow reusing parts of other models, which were not designed for modularity, it also introduces a few caveats. These arise when considering constraints, which exist outside and independent of the feature tree. As constraints can reference any features in a model, when only a subtree of that model is referenced somewhere, there could still be constraints that reference other features. When a constraint uses only features from within that subtree, it is clear that it should also be contained in the referencing model. However, when it references a feature from the subtree and also other features, those features would be outside of the feature tree for the referencing model. Thus, these out-of-tree features would not be visible in the normal tree representation, but would still have to be considered, as they could have an impact on the configuration space. Multiple features in the subtree could also be affected transitively through a chain of constraints,

https://doi.org/10.24355/dbbs.084-202008211051-0

```
imports
  org.featuremodels.FileSystem as fs // import from namespace
  OperatingSystem as os refer [Windows, macOS, Debian]
  // Using refer features can be used without a prefix.
  // 'refer all' is also possible, but may cause name clashes.
features
  Server
    mandatory
      fs.FileSystem
      os.OperatingSystem
    optional
      Logging

constraints
  Windows => fs.NTFS
  macOS => fs.APFS
  Debian => fs.EXT4
```

Listing 4.5: Example composition of the submodels to construct the entire server.

possibly even including constraints that only consider out-of-tree features. Hence, analyzing which constraints have to be considered and which are safe to drop is not trivial (cf. [ACL+11; SKT+16]). When it was allowed to only include subtrees, tools using the language would need to handle these out-of-tree features to correctly analyze and evaluate constraints. For tools that natively support similar semantics of multiple models, implementing this might not be very difficult. Other tools might support the implementation using hidden features and multiple root features, but this would come with its own problems and would add considerable complexity for the implementer. This would certainly hinder adoption.

An alternative approach is to include any constraints that only include features from the referenced subtree. These constraints are assumed to specify the internal relationships of that subsystem. Constraints that use at least one other feature could be thought of as describing the usage of that subtree in the context of the larger system. This information could be assumed to be relevant only to that specific larger system, so they could be ignored for any other referencing model. This way, constraints could be filtered with a simple predicate that just checks inclusion in the subtree for each feature that is part of the constraint. While these assumptions might be accurate for some systems, they certainly do not hold for every system. Since some constraints that could be important even for the subtree could be dropped, leading to a loss of information, this is a dangerous assumption to make.

When it is allowed to include only a subtree of an imported model, handling constraints becomes either very complex or makes assumptions about the structure of the imported model. Thus, we only allow referencing the root feature of a submodel in the feature tree. This way, the whole tree and all constraints can be considered, greatly reducing the integration complexity for tool vendors.

Another decision to make for references is whether it should be allowed to reference the

https://doi.org/10.24355/dbbs.084-202008211051-0

same model multiple times under different aliases. This could be allowed when using the *as* keyword. An example use case for this is a system with hardware components that are instantiated multiple times but configured differently. This could be the case in a car that has multiple different seats or a software-intensive system that runs on multiple different servers. For instance, a web service could include a submodel of a database and alias it multiple times for each continent. Then, the *refer* keyword cannot be used for these namespaces, as there would be no way to know which of the instances it would refer to in the tree. However, using only aliasing and not the *refer* keyword, the translation to feature models without the support for imports would be straightforward. Through renaming of the included features by prepending each feature's name in the submodel with the alias, every instance of each feature can be distinguished clearly. In the example, there might be a `us.Database` and a `europe.Database`, with their children also renamed to `us.Logging`, etc. Allowing multiple instances could mean additional complexity for tool vendors implementing the language. Still, it could be worthwhile for the increased flexibility for modeling systems.

4.5. Syntax of Constraints

Constraints are formulas in propositional logic. This expressiveness was voted for most often (cf. Figure 3.8) and it is usually efficient to analyze using SAT solvers, although exceptions exist (cf. [BTS19]). Furthermore, it is expressive enough to represent most real-world cases [KTM+17]. We define the syntax of our formulas in the following way: A formula is either an atomic formula or a complex formula. Atomic formulas can be `true`, `false`, or references to features. Let `F` and `G` be two (not necessarily different) formulas. Then the following constructions are complex formulas and thus also formulas:

- A formula that is wrapped in parentheses
- A negated formula: `!F`
- The conjunction: `F & G`
- The disjunction: `F | G`
- The implication: `F => G`
- The equivalence: `F <=> G`

Any ambiguities are resolved according to precedence rules. The precedence rules of the operators from highest to the lowest are: `()`, `!`, `&`, `|`, `=>`, `<=>`. To illustrate the precedence rules, consider the example formula `!A & B <=> C | D => E` without any parentheses. This formula is equivalent to `((!A) & B) <=> ((C | D) => E)`. Even though the representation with parentheses around every formula reduces the chance of misinterpretation, we still allow the representation relying on precedence rules. It is shorter and more readable in many cases.

The reason we choose symbols such as & instead of the keyword *and* is that they are shorter, but still convey the meaning. Using & for an *and*, the pipe symbol | for an *or*, and the exclamation mark ! for negation is common practice in most programming languages, so it is familiar to

https://doi.org/10.24355/dbbs.084-202008211051-0

many users. Also, it prevents the overloading of the keyword *or* that is also used for the *or* group in the feature tree.

4.6. Summary of the Proposed Concepts

In summary, we present two different concepts for variability languages, a composition mechanism, and a syntax for constraints. Common to both languages is the concept of attributes to specify additional data and the effort to minimize the number of keywords to write while still maintaining readability. We describe the common characteristics and reasoning behind the languages in more detail in Section 4.1. The first language concept uses nesting and indentation (cf. Section 4.2), while the second adopts references to represent the hierarchy (cf. Section 4.3). For both languages, we reuse the syntax for references to enable composition (cf. Section 4.4) and for constraints (cf. Section 4.5), which use propositional logic.

Still left to present are grammars to formally describe the languages, along with tool support including parser and printer to ease the integration into existing variability tools. We choose a suitable implementation technology for tool support, introduce grammars, and describe the developed tool support in Chapter 5. Also, the approaches need to be evaluated by questioning the community and revisiting the requirements. We discuss the results of the evaluation in Chapter 6.

https://doi.org/10.24355/dbbs.084-202008211051-0

https://doi.org/10.24355/dbbs.084-202008211051-0

5. Tool Support for UVL

A language without tool support cannot be used in practice. Thus, we provide a default library that can be integrated into existing SPL tools. In this chapter, we elaborate on how we choose the parser technology for the library (Section 5.1). We present grammars for both language concepts (Section 5.2) and a default implementation of the parser library (Section 5.3). Finally, we demonstrate its usability by integrating it into the FeatureIDE tool (Section 5.4).

5.1. Choosing a Parser Library

A vast number of different parser libraries and generators are available to choose from. They all serve the same basic purpose, providing an implementation of a parser for a supplied grammar. Hence, we have to carefully weigh requirements and criteria to derive a decision. In Section 3.5, we established that the most important criterion for the tool support is the ease of integration of a parser and printer into existing variability tools. Thus, we focus on providing a small parser without many dependencies.

Prerequisites for a parser library are that it uses standard EBNF notation or a slight modification thereof. This way, the grammar itself can be reused in case the need arises to ever switch the parser library or implement alternative libraries in other programming languages. The parser library should also support free-text parsing instead of generating projectional editors. To limit the implementation effort, also a scanner should be generated or included. The scanner should be customizable to allow the handling of indentation-aware languages. These factors already limit the pool of available libraries considerably.

Since most variability tools are written in Java (e.g., pure:variants [pur20], FeatureIDE [MTS+17], DOPLER [DRG+07], mbeddr [KVR+20], AHEAD [Bat05]), the community would benefit most from a library that can be used on the Java Virtual Machine (JVM). Nonetheless, the option to generate parsers in additional languages could be beneficial for other tool vendors that do not use Java. A nice-to-have feature would be the option to provide IDE integration and editor support. However, this should not be at the expense of making the integration more complex, requiring more dependencies, and a heavier runtime library.

A lightweight option to provide editor support, without relying on any specific IDE could be the Language Server Protocol (LSP) [Mic20]. It is a protocol that can significantly reduce the effort required to integrate language support into different editors or IDEs [Bün19]. It is widely supported in all the major IDEs and extensible text editors today. The editor is an LSP client, sending the text and information about actions the user performs to the language server. The server then answers with information about syntax highlighting, annotations, problems, and completions.

There are not many comparisons of different language workbenches or parser generators

https://doi.org/10.24355/dbbs.084-202008211051-0

available. This is because comparing these tools meaningfully requires significant effort and even then often cannot take into account the various specialties of the different tools. To make a comparison, a set of predefined tasks would have to be solved with different tools. Then the time it took to implement those solutions as well as the quality of the solutions themselves would have to be evaluated. To make it comparable, these tasks should be solved by multiple independent persons or teams. When there are not enough participants, each participant would have to complete the tasks with multiple different tools. The order of the tools used has to be varied for each participant to prevent interference of learning effects with the results. Even with these measures, it would only shed a light on how well these tools are suited to solve a task initially. They could not make accurate predictions on how maintainable the solutions would be in the long run. That said, there are papers comparing a few tools for a single problem (e.g., [CPB+07], [PP08], or [Mer10]), but they generally have very different scopes and criteria. This way, they cannot be used to aggregate the data into a good overview of all the different approaches.

One effort trying to overcome these issues existed, namely the language workbench challenge [BEH+17]. The idea was that there would be a yearly challenge, language engineers could solve using different language workbenches. Challenges were held from the year 2011 to 2014 and then once again in 2016, but the project has been discontinued since and the associated domain has been sold. From these challenges, only the one from the year 2013 was designed in a way that led to comparable results, which are summarized by Erdweg et al. [EvdSV+15].

In Table 5.1, we give an overview of a selection of parser generators and language workbenches. ANTLR4 [ANT20b] is one of the most popular parser generators written in Java, but targeting many other languages besides Java. It provides extensive documentation and a repository with over 180 example language implementations in the form of ANTLR4 grammars [ANT20a]. It is a plain parser and scanner generator, generating an adaptive LL(*) parser (cf. [PHF14]). For the grammar input, it uses its own format that mixes grammar aspects and implementation aspects. This way, a parser can be generated from a single input file, but that file itself is specific to ANTLR4, although based on EBNF. ANTLR3 is also still a popular option using a different grammar format that is not compatible with ANTLR4 and generates an LL(*) parser. Coco/R [MLW18] has a similar approach but can generate parsers in even more languages, although the available target languages depend on the development platform used. It does not seem to be maintained and developed as actively as ANTLR4. JavaCC [Jav18] is another similar option. It can generate code in the three target languages Java, C#, and C++.

A different approach is taken by instaparse [Mar19]. It aims to be the simplest way to build parsers in Clojure. Clojure is a modern LISP targeting the JVM, so it can both use Java libraries and provide libraries on its own. Instead of generating static Java code for a given grammar in advance, it intends to be more dynamic, allowing us to generate that code during runtime. For this, it uses the dynamic metaprogramming facilities of the host language Clojure. This way, it is very flexible and allows to iterate any context-free grammar quickly.

Then there are different language workbenches. Rascal, Spoofax, SugarJ, and Xtext are a selection of the available options. From those workbenches, SugarJ [EKR+11] supports the

https://doi.org/10.24355/dbbs.084-202008211051-0

Tool	IDE support	Target Languages	LSP	Documentation	Maintained
ANTLR4	no	8	no	good	yes
Coco/R	no	14	no	good	1 yr. ago
JavaCC	no	3	no	good	yes
instaparse	no	Clojure/JVM	no	good	1 yr. ago
Rascal	yes	Java	no	sparse	yes
Spoofax	yes	Java	no	good	yes
SugarJ	yes	3	no	sparse	4 yr. ago
Xtext	yes	Java	yes	sparse	yes

Table 5.1.: Overview of characteristics of selected parser generators and language workbenches.

most target languages but does not seem to be maintained any longer. Rascal [Cen14] is a metaprogramming language that can be used to specify DSLs, but its documentation is lacking. Xtext [Ecl20] is probably the most well-known and widely-used language workbench in the Eclipse ecosystem. A unique feature of it is that it can generate an implementation for the LSP in addition to the parser and editing facilities for IDEs. Internally it uses ANTLR3 as its parser generator. Spoofax [Met20] is a well-documented alternative to Xtext with its own parser generator. It is an academic product that is not used as widely as Xtext.

The language workbenches are huge frameworks requiring substantial effort to familiarize oneself with their internals to get started and customize their workings beyond simple hello-world languages. Thus, we rather choose one of the smaller, more narrowly focused parser generators. From these, choosing the best one is not as important as choosing one with a good fit that can quickly be swapped for another library at a later stage. The simplest to get started with and also the most flexible of the presented options seems to be instaparse. So we choose it for the first implementation of a small default library and to iterate the grammars themselves.

5.2. Grammars for the Languages

Listing 5.1 shows a grammar that can be used to parse the language presented in Section 4.2. It uses an EBNF-like notation. Rules surrounded by angle brackets are hidden, so they are important for parsing the structure, but no longer in the final AST. Strings prepended with a '#' are regular expressions, as they would be recognized by the Java regex utilities.

The language is indentation-aware, which is not directly supported by classical parsers. Thus, the scanner has to be modified to emit indent and dedent tokens whenever a section that is more or less indented starts. The exact semantics for this process is as follows: First, we distinguish between physical and logical lines. Physical lines are actual lines in the text that are separated by the usual linefeed or carriage return characters. Logical lines can be comprised of multiple physical lines when they are to be joined implicitly or explicitly. Implicit joins occur within environments that are surrounded by parentheses, brackets, or curly braces, where splitting

https://doi.org/10.24355/dbbs.084-202008211051-0

long lists of items on to multiple lines is allowed to increase the readability. Explicit joins occur, whenever a physical line ends with a backslash, mimicking the behavior of languages such as Python or bash. Second, line comments are ignored by the scanner and do not cause it to emit any tokens until the end of a physical line is read. Third, we require a stack storing the previously encountered levels of indentation. For each new logical line, the level of indentation is compared to the previous level on the top of the stack. When it is indented more, an indent token has to be emitted and the new level is pushed to the stack. If it is indented less, the level has to be compared to previously seen levels of indentation, to see which block is continued. For each level that is popped off the stack, a dedent token has to be emitted, until the matching level with the same indentation is found on the top of the stack. When no matching level of indentation can be found, the line is indented to a wrong level. This is an indentation error, as it is unclear to which block the code on the line is supposed to belong. Empty lines or lines that consist only of whitespace characters are ignored for this consideration. Indentation can consist of spaces or tabs. When a mixture of spaces and tabs is used that makes the comparison to previous levels ambiguous, it also causes an indentation error.

The grammar for the referencing language concept is similar in the overall structure and uses the same rules as the nesting concept for its constraints. One critical difference is that it relies more on line breaks to be unambiguous. Each new feature specification has to start on its own line, so the parser should not ignore line-breaking characters, as they are explicitly part of the grammar. All other whitespace between tokens should still be ignored, so the grammar does not have to declare every position where whitespace could occur explicitly. Listing 5.2 shows the grammar for the referencing language concept.

5.3. UVL Parser Library

We provide a default implementation for the language in the form of a small parser library. The library is released under the MIT license and the source code is available on GitHub.[1] It provides an API to read some text in UVL and return a parsed AST. Furthermore, there is an implementation of a printer to generate UVL files programmatically. We implement the library using the nesting language concept, but the grammar could quickly be swapped out. This would require a few internal changes to the way the AST is constructed, but no changes to the external API should be required.

The library is designed to require little dependencies and can be distributed in a single jar file. To parse a text file, there exists a single static method *parse* that takes the text as an input, as well as a file loader to resolve any imports in the UVL file. Since the path to project files or the way to access them might be different for each consumer of the library, the file loader is a callback method that can be implemented in the consuming variability tool. It is expected to take the namespace identifier string as an input and return the text of the corresponding file. For clients that have no special way of loading files, a default implementation is supplied that will load files relative to a path given as a string to the parse method. When imports are not used

[1] UVL library on GitHub: `https://github.com/neominik/uvl-parser`

https://doi.org/10.24355/dbbs.084-202008211051-0

at all by the client's UVL files, the argument can be omitted entirely. We do not recommend using this arity of the parse function in production, as the parser will probably be unable to find the correct files to load, in case a UVL file uses imports. The return value of the parse function is either the parsed AST in the form of an object of the class *UVLModel* or when there were parse errors, a *ParseError*. Figure 5.1 shows the structure of the returned AST and *ParseError* as a UML class diagram.

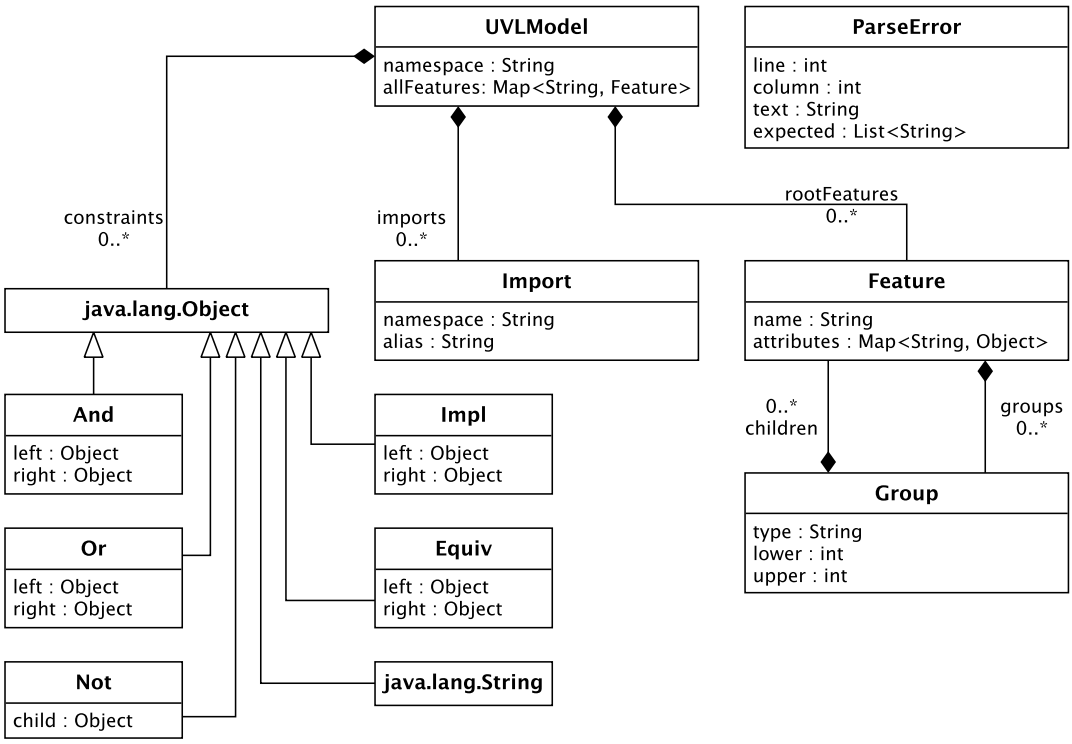


Figure 5.1.: UML class diagram of the AST package. For brevity, we show the fields of the classes only, although they are private. Access is only possible through getters and setters in the JavaBeans style.

A *UVLModel* contains the namespace, imports, root features, constraints, and a map with all features in all submodels, indexed with their namespace-prefixed name. This last map can be used for constant-time resolution of any feature when it is referenced by name, or to retrieve the set of all the features. Features contain groups, which in turn contain the children of the features. Groups have a type property, which can hold the values 'or', 'alternative', 'optional', 'mandatory', or 'cardinality'. In the latter case, the values for the lower and upper bounds have to be considered. Constraints can be of the types *And*, *Or*, *Impl*, *Equiv*, *Not*, or *String*. The first four have two children each, a left one and a right one, while *Not* has only one child. Strings are used as literal references to other features and are always the leaf nodes of the constraint's AST. Should parsing fail because the provided text did not comply with the UVL grammar, a *ParseError* is returned by the parse function. It indicates what went wrong, the location in the

https://doi.org/10.24355/dbbs.084-202008211051-0

file, and alternatives the parser would expect at that position.

The printing functionality of the library is realized in the *toString* methods of the objects. Calling *toString* on any object returned by the parse method or any descendants in the AST produces the representation of the object in UVL. Only the printed output of the root element *UVLModel* produces a complete and valid UVL file ready for storage or exchange with other tools.

5.4. Integration into FeatureIDE

To demonstrate the usability of the library, we integrated it into the variability tool FeatureIDE.[2] We choose FeatureIDE because is one of the most widely used open-source tools for feature modeling [BRN+13] and developed in part at our university. In FeatureIDE, the library is included as a single jar file. To adhere to the way other exchange formats are implemented, we create a new format. It calls the parse method of the library and constructs the internal model representation of FeatureIDE by iterating through the returned AST. Similarly, for printing, we iterate through the internal model and construct an AST using the classes of the library. Finally, we call *toString* once on the root element to produce the text that can then be used to store the file.

[2] The functionality is scheduled to be released in the next major release (3.7.0 or 4.0.0). Link to the fork with the corresponding changes on GitHub: `https://github.com/neominik/FeatureIDE`

https://doi.org/10.24355/dbbs.084-202008211051-0

```
FeatureModel = Ns? Imports? Features? Constraints?

Ns = <'namespace'> REF
Imports = <'imports'> (<indent> Import+ <dedent>)?
Import = REF (<'as'> ID)? (<'refer'> Refer)?
Refer = (<'['> (ID <','?>)* <']'>) | 'all'

Features = <'features'> Children?
<Children> = <indent> FeatureSpec+ <dedent>
FeatureSpec = REF Attributes? Groups?
Attributes = (<'{'> <'}'>) | (<'{'> Attribute (<','> Attribute)* <'}'>)
Attribute = Key Value?
Key = ID
Value = Boolean | Number | String | Attributes | Vector | Constraint
Boolean = 'true' | 'false'
Number = #'[+-]?(0|[1-9]\d*)(\.\d*)?([eE][+-]?\d+)?'
String = #'"(?:[^"\\\n]|\\.)*"'
Vector = <'['> (Value <','?>)* <']'>
Groups = <indent> Group* <dedent>
Group = ('or' | 'alternative' | 'mandatory' | 'optional' | Cardinality)
        Children?
Cardinality = <'['> (int <'..'>)? (int|'*') <']'>

Constraints = <'constraints'> (<indent> Constraint+ <dedent>)?
<Constraint> = disj-impl | Equiv
Equiv = Constraint <'<=>'> disj-impl
<disj-impl> = disj | Impl
Impl = disj-impl <'=>'> disj
<disj> = conj | Or
Or = disj <'|'> conj
<conj> = term-not | And
And = conj <'&'> term-not
<term-not> = term | Not
Not = <'!'> term
<term> = REF | <'('> Constraint <')'>

indent = '_INDENT_'
dedent = '_DEDENT_'
<strictID> = #'(?!alternative|or|features|constraints|true|false|as
                |refer)[a-zA-Z][a-zA-Z_0-9]*'
<ID> = #'(?!true|false)[a-zA-Z][a-zA-Z_0-9]*'
REF = (ID <'.'>)* strictID
<int> = #'0|[1-9]\d*'
```

Listing 5.1: Grammar for the nesting language concept in EBNF-like notation. All whitespace characters should be ignored automatically.

https://doi.org/10.24355/dbbs.084-202008211051-0

```
FeatureModel = Ns? Imports? Features Constraints?

Ns = <'namespace'> REF
Imports = <#'\n?imports'> Import*
Import = <'\n'> REF (<'as'> As)? (<'refer'> Refer)?
As = ID
Refer = (<'['> (ID <','?>)* <']'>) | 'all'

Features = <#'\n?features'> FeatureSpec*
FeatureSpec = <'\n'> strictID Children? Attributes?
Children = <':'> Group? Child*
Group = 'or' | 'alternative' | Cardinality
Cardinality = <'['> (int <'..'>)? (int|'*') <']'>
Child = REF '!'?
Attributes = (<','> Attribute)+
             | (<','> Attribute)* <indent> (<'\n'> Attribute)* <dedent>
Attribute = Key Value?
Key = ID
Value = Boolean | Number | String | Map | Vector | Constraint
Boolean = 'true' | 'false'
Number = #'[+-]?(0|[1-9]\d*)(\.\d*)?([eE][+-]?\d+)?'
String = <'"'> #'(?:[^"\\\n]|\\.)*' <'"'>
Map = <'{'> (Attribute <','?>)* <'}'>
Vector = <'['> (Value <','?>)* <']'>

Constraints = <#'\n?constraints'> (<'\n'> Constraint)*
Constraint = disj-impl | Equiv
Equiv = Constraint <'<=>'> disj-impl
<disj-impl> = disj | Impl
Impl = disj-impl <'=>'> disj
<disj> = conj | Or
Or = disj <'|'> conj
<conj> = term-not | And
And = conj <'&'> term-not
<term-not> = term | Not
Not = <'!'> term
<term> = REF | <'('> Constraint <')'>

indent = '\n_INDENT_'
dedent = '\n_DEDENT_'
strictID = #'(?!alternative|or|features|constraints|true|false)
              [a-zA-Z][a-zA-Z_0-9]*'
ID = #'(?!true|false)[a-zA-Z][a-zA-Z_0-9]*'
REF = (ID <'.'>)* strictID
```

Listing 5.2: Grammar for the referencing language concept in EBNF-like notation. All whitespace characters that are not line breaks should be ignored automatically.

https://doi.org/10.24355/dbbs.084-202008211051-0

6. Evaluation

After proposing two language concepts and developing tool support for the first, we evaluate the language concepts in this chapter. For this, we conduct a second questionnaire (Section 6.1), revisit the requirements (Section 6.2), and evaluate the scalability of the language (Section 6.3).

6.1. Second Questionnaire

To gather the community's thoughts and feedback about the proposed language concepts and determine which one is preferred, we design and conduct a second questionnaire. In the following section, we describe its structure and questions, before discussing the results in Section 6.1.2. Due to time restrictions, we cannot conduct this questionnaire at the next MODEVAR event but have to distribute it via the MODEVAR mailing list. Thus, this time participants work on the questionnaire individually, instead of in pairs. As teaching and learning are considered relevant and useful (cf. Section 3.2), we also distribute the questionnaire to students who are currently enrolled in courses on SPLs at TU Braunschweig and University of Ulm.

6.1.1. Overview of the Second Questionnaire

The goal of the second questionnaire is to determine the community's opinion on both language concepts and which one is preferred by most experts. The survey starts by introducing the server example (cf. Figure 2.2) as a feature model. Participants can rely on that representation as a reference to understand the semantics of the proposed languages. We ask for the size of the largest variability model the participants have ever worked on and for the size of typical models. This is to put the given answers into context.

Then, we present the nesting language concept, similar to our presentation in Section 4.2. Participants are asked to rate how much they like the concept on a scale from 1 (not at all) to 6 (very much). To get more detailed results on what they like or dislike specifically, we ask about the level of agreement to the following statements:

- The style is good.
- It is well suited for teaching and learning.
- It is too complex.
- It can easily be integrated into my tool.
- I can represent my models in this language.

https://doi.org/10.24355/dbbs.084-202008211051-0

We ask what the participants would change about the language concept and how happy they would be with the language when those changes were applied. Similarly, we present the referencing concept, as in Section 4.3. To keep the answers for both language concepts comparable, we ask the same questions about their rating, agreement, and change requests.

As in Section 4.4, we present the composition mechanisms only once for both language concepts. Here, we ask the same questions as before, with the addition of one further question: Using namespace aliasing with *as*, the same feature model could be included multiple times under different aliases in the same file. We ask whether the participants deem this functionality useful and would include it or disallow it. Including it might add more complexity to the implementations while disallowing it would be more restrictive. Thus, we ask for their reasoning behind the given answer.

After introducing both language concepts and the composition mechanism, we ask them to choose the preferred language concept of the two. We ask for the reasons behind their decision, as well as any reluctance they might have with their answer. Also, as the goal is to find a language the community can agree on, we ask whether they would still agree to the other option, should the majority of the community vote for that one. In case they vote for *no*, we ask what should be changed in that language to gain their acceptance. Finally, we ask for further comments, requests, or feedback. For reference, we again include the list of all questions in the appendix (cf. Appendix A.2).

6.1.2. Results of the Second Community Questionnaire

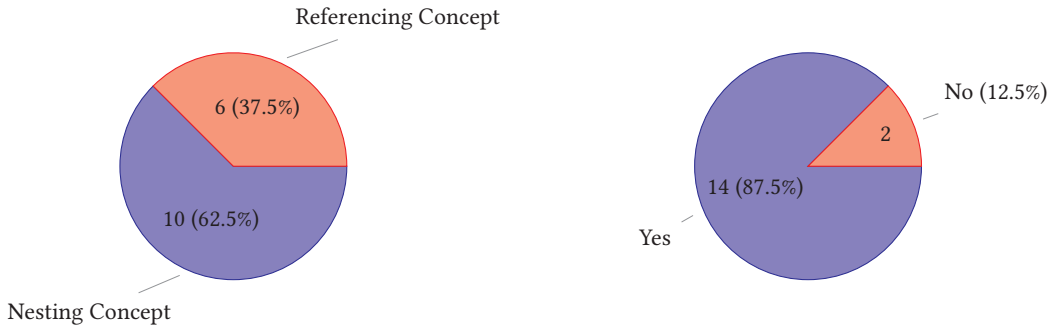
In this section, we present and discuss the results of the second questionnaire, as answered by the SPL community. For this, we first focus on general observations and the result which language has been voted for most often, before turning to the individual concepts. We discuss the results of the students' answers in Section 6.1.3.

General Observations and Language Choice

16 participants submitted their answers to the questionnaire. Overall, the nesting language concept has been voted for more often (see Figure 6.1a). Nine participants preferred the nesting one, while only six voted for the referencing language. However, two participants who voted for the referencing language said that they do not have a strong preference, but find the standardization more important than the actual syntax. When asked if they would also agree to the other language, should the majority vote for it, most participants chose yes (see Figure 6.1b). Only two voted for no. Incidentally, of those two votes, there is one for each language.

Asking for the size of typical models participants work on and the largest models they have worked on helps to better put the answers into context. Especially, when regarding concerns of scalability for the languages. In Figure 6.2, we show the distribution of their answers. Most participants typically only work on small models, between ten and a hundred features in size. None of the participants typically works on models that contain more than ten thousand features. However, the largest models they have ever worked on are considerably larger than the typical

https://doi.org/10.24355/dbbs.084-202008211051-0

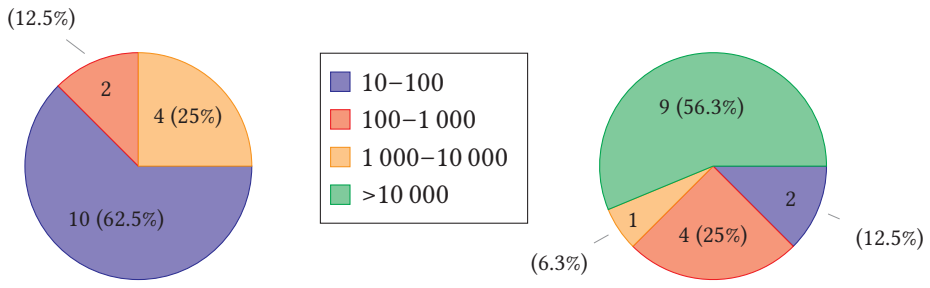


(a) Preferred Language (b) Accept Majority Vote

Figure 6.1.: Voting results on which language concept is preferred (a) and whether participants would also agree to the other language, should the majority vote for that option (b).

ones for most participants. More than half of the participants have worked on a model that was larger than ten thousand features.

In the first questionnaire, as well as in discussions on the last MODEVAR workshop, most people argued that the referencing language would be better suited for large models. Their main argument was that using references for the hierarchy limits the amount of nesting and indentation. Thus, we expected the people voting for the referencing language to work on larger models than those who vote for the nesting one. However, as shown in Figure 6.3, in our sample, the opposite seems to be the case. While participants voting for the nesting concept tend to work on larger models only slightly more often, the difference is more pronounced in the size of the largest models. A third of the voters for the referencing concept claims to have never worked on a model larger than 100 features. In our sample, voters for the nesting concept consistently have worked on larger models. 70% even have worked on models with more than 10 000 features, compared to only one-third of the participants voting for the referencing language. However, with only 16 participants, these statistical considerations might not be representative. Still, in our sample, participants working on large models vote for the nesting language concept more often, despite alleged scalability issues. This is evidence that the syntax is less important when dealing with large models than a mechanism for composition and decomposition.



(a) Typical Models (b) Largest Models

Figure 6.2.: Number of features of (a) typical and (b) largest feature models participants work on.

https://doi.org/10.24355/dbbs.084-202008211051-0

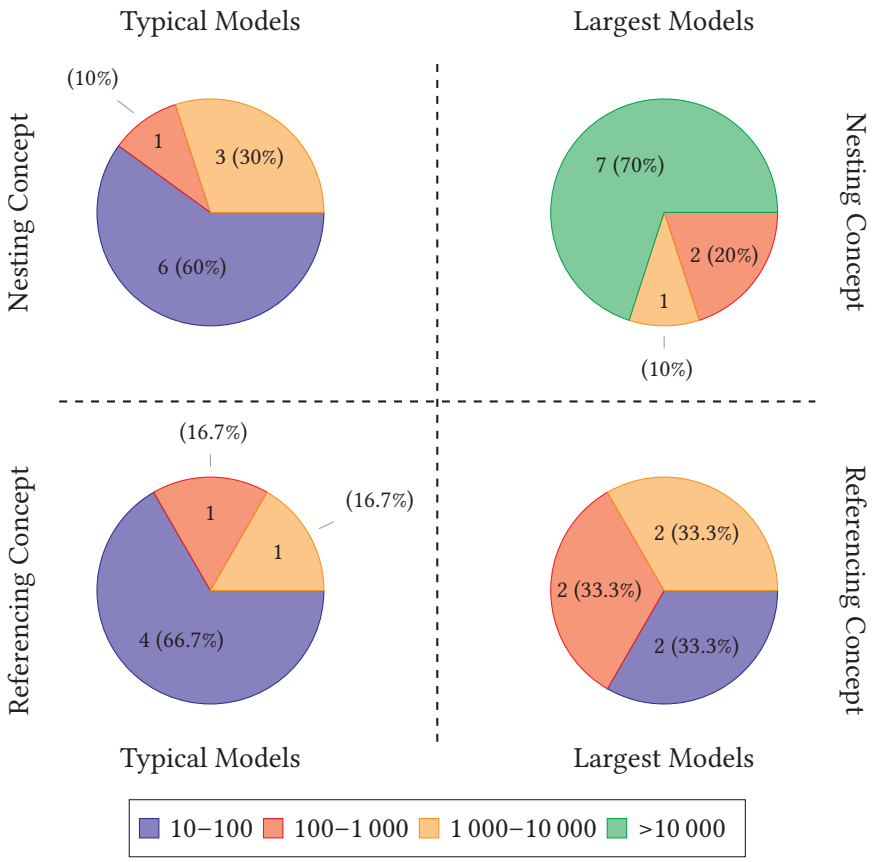


Figure 6.3.: Number of features of typical and largest feature models participants work on. Grouped by their preferred language.

Detailed Results for the Nesting Language Concept

Overall, the nesting language concept is preferred over the referencing one. In Figure 6.4, we show the participants' overall contentment with the language colored in blue. Most participants rated it as five out of six, with six being the best grade and one the worst. Only two answers rated it in the lower half of the spectrum. Additionally, we asked what they would change about the language and how they would like the language when those changes were applied. We show those results in red in Figure 6.4. After their suggested changes, the overall rating improved slightly, with now four participants giving the best grade and only one falling in the lower half with a three out of six.

Two participants suggested that the group keyword should be repeated for each feature, instead of only once per group. They fear that with many features it could become more difficult to see which group a specific feature belongs to. Rick Rabiser states, "For deeper hierarchies, indentation alone might not be sufficient – I suggest to allow to (also) use some form of brackets + separators, at least for feature groups." Although introducing additional brackets might help slightly with keeping an overview, syntax alone can not help much with large models. That is why we also introduce a mechanism for the decomposition of large models. At this point in the

https://doi.org/10.24355/dbbs.084-202008211051-0

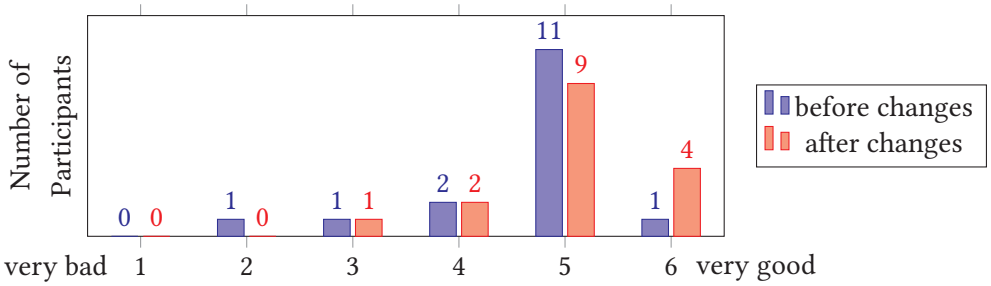


Figure 6.4.: Voting results on how well the nesting language concept is liked before and after suggested changes.

questionnaire, however, participants were not yet introduced to that concept. An argument against additional brackets is that for the implementation complexity of a parser as well as for teaching, having multiple syntaxes to specify hierarchies should be avoided.

For more detailed feedback on what participants like or dislike specifically, we asked for their agreement to the statements introduced in Section 6.1.1. We show the results in Figure 6.5. 15 out of 16 participants agree that the style is good. Also, the majority is of the impression that this language would be well suited for teaching and learning. Most participants strongly disagree with the language being too complex. Finally, most participants agree that the language could be integrated easily into their tools and that they could represent their models with it.

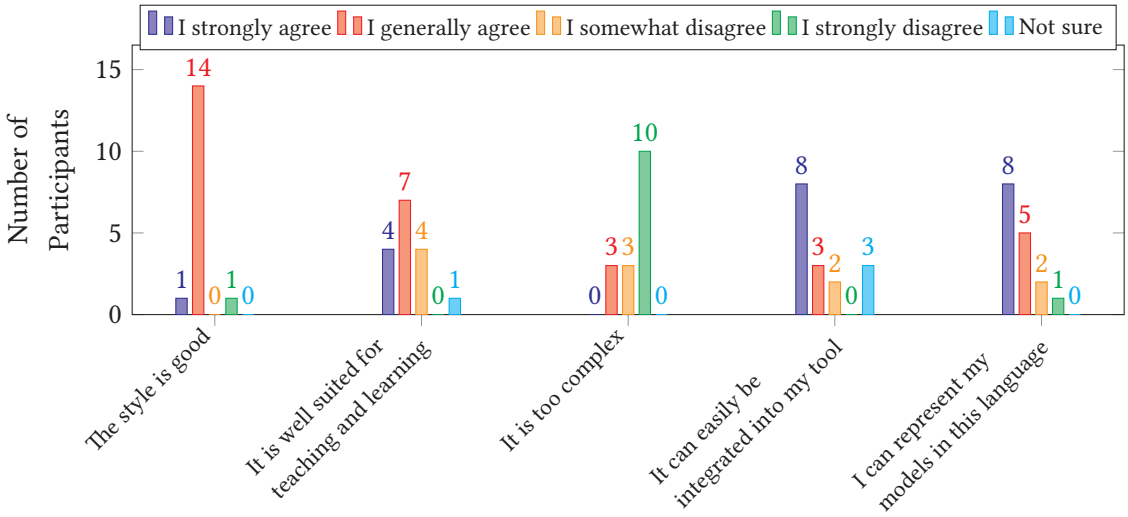


Figure 6.5.: Voting results on the level of agreement to the given statements about the nesting language concept.

Detailed Results for the Referencing Language Concept

As the referencing concept was preferred less often than the nesting one, it is not surprising that the overall rating is also less positive. In Figure 6.6, we show how well the participants rated

https://doi.org/10.24355/dbbs.084-202008211051-0

that concept before and after their suggestions for improvement. In both cases, the number of participants is distributed almost equally in both the lower and the upper half of the rating. The result is even slightly worse after their suggestions for changes. Paul Bittner said, "Since attributes belong to a feature and not to that feature's children, I would like to have the attributes before the colon and not integrated into the list of children. [...] But maybe it is sufficient to use newlines as already supported." Sebastian Kreiter has a similar opinion: "I would always put the attributes on a separate line as it was done for the feature Logging. This would get rid of the confusing comma and avoid that attributes are overlooked." Indeed, it might be confusing that the attributes of the parent appear after the list of children in the line. If that list is long, it might be difficult to associate the attributes to the corresponding feature. A better solution might be to drop this notation or allow attributes to appear before the children, as suggested by the participants. However, a long list of attributes should always be specified in separate lines. Otherwise, the same problem arises in reverse, with the children suddenly being far away from the parent. Furthermore, Sebastian Kreiter would prefer "to also use the keywords optional and mandatory in this language rather than '!'. This would make the language more consistent and easier to read in my eyes." This is the same reasoning as in the nesting language concept, only applied to this one. Grouping optional and mandatory features together in this way would differ more from the existing languages, which specify optionality on a per-feature basis. However, as Sebastian Kreiter said, it would be more consistent, since the group type would be specified only between the parent and the features, instead of sometimes in-between and other times at the children.

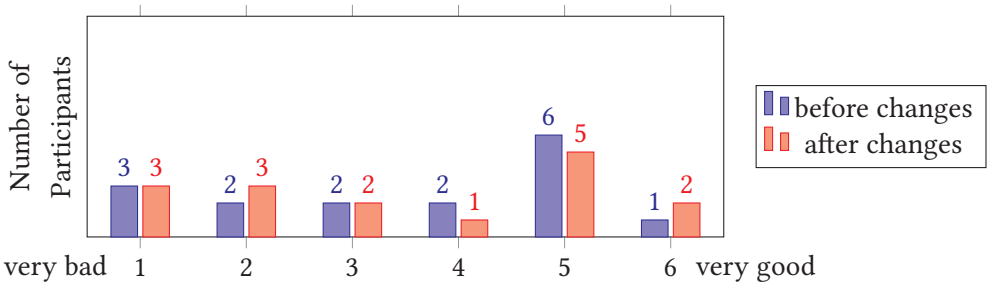


Figure 6.6.: Voting results on how well the referencing language concept is liked before and after suggested changes.

Just as the overall satisfaction, the more detailed agreements to the individual statements are worse than for the nesting concept (see Figure 6.7). The agreement about good style is only around half as present as before. Only eight participants agree with the statement, while the other half disagrees somewhat or strongly. 11 of the 16 participants think that it is not well suited for teaching and learning, although most still disagree with the statement that it is too complex. While there is a majority thinking that the language could be integrated into their tools easily, the level of agreement decreased slightly compared to the nesting concept. Still, the voting results regarding whether participants can represent their models in the language are about the same as before. 14 out of 16 agree that they can represent their models in the

https://doi.org/10.24355/dbbs.084-202008211051-0

language. This is not surprising, since both languages have the same language features, only a different concrete syntax, so the same models can be represented. However, it is reassuring that this fact also translates to the votes of the community.

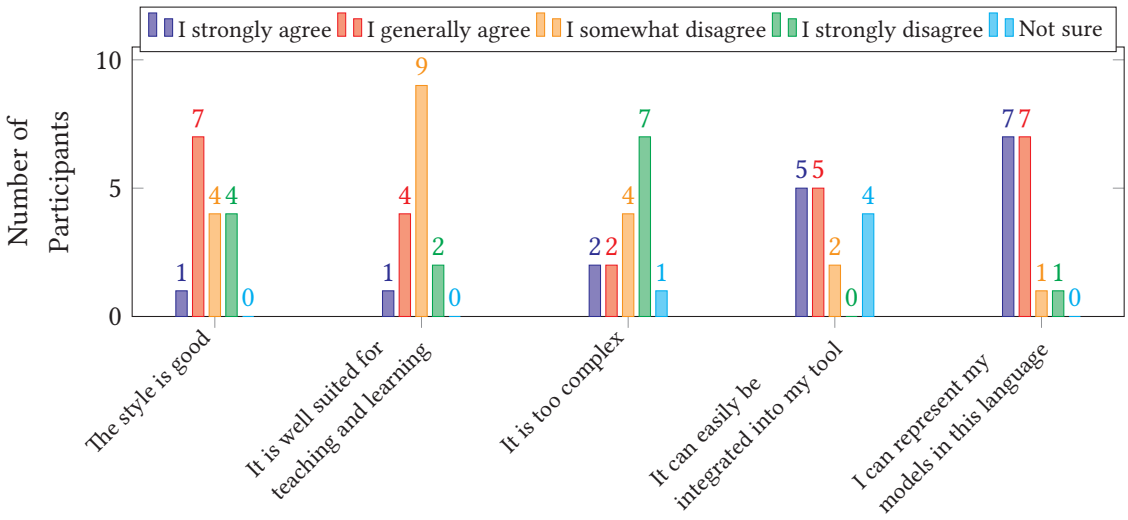


Figure 6.7.: Voting results on the level of agreement to the given statements about the referencing language concept.

Detailed Results for the Composition Concept

Overall, the composition concept is well received by the community. In Figure 6.8, we show the rating for the concept. While one participant gave the concept the worst rating, 10 of 16 rated it as good or very good before their suggestions and 11 of 16 after their suggestions. The most common comment for the concept is that it is too complex. Two participants suggest removing the *refer* keyword to make the language simpler and avoid name clashes. The *refer* concept is intended to simplify manually writing UVL files but adds complex semantics. These make both reading files and resolving references in the parser more difficult. We made the same observation during the implementation of the tool support. Thus, we agree that it is a good idea to drop that concept to simplify the language and the implementation. Consequently, we do not implement the *refer* concept in our tool support.

In the detailed rating, we also see that complexity is the most criticized point for the composition mechanism (see Figure 6.9). 7 of 16 participants agree with the statement that the concept is too complex, which is almost half of the participants. The other statements are rated more positively. 11 of 16 participants agree that the style is good and that the concept is well suited for teaching and learning. Half of the participants agree that it can be integrated into their tools easily. 25% disagree and the remaining 25% are not sure. However, 14 of 16 participants agree that they can represent their models with this concept.

On whether to allow including the same model multiple times with different aliases, 10 of 16 voted to allow multiple instances (see Figure 6.10). Participants who voted in favor of

https://doi.org/10.24355/dbbs.084-202008211051-0

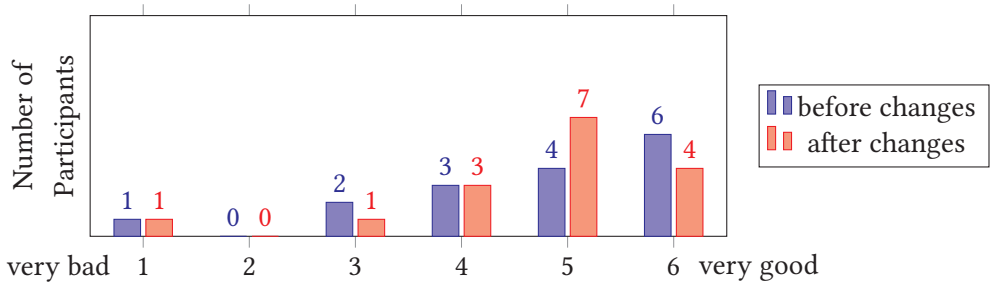


Figure 6.8.: Voting results on how well the composition mechanism is liked before and after suggested changes.

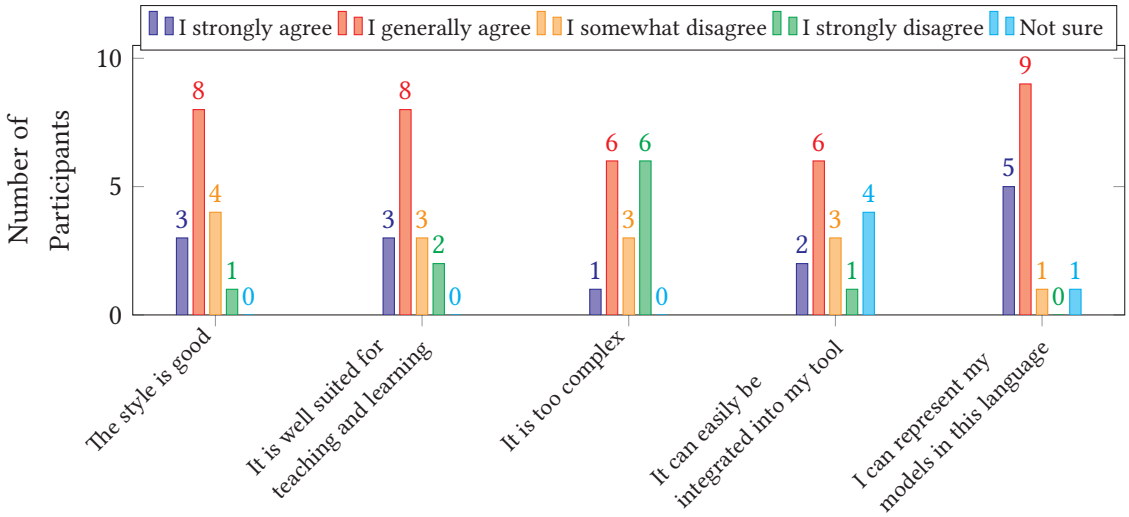


Figure 6.9.: Voting results on the level of agreement to the given statements about the composition mechanism.

multiple-instantiation argue that this language feature has the potential to reduce redundancies in real-world models. Some participants are actively working with models where this feature is required. They say that there is no high cost for integrating it. In contrast, participants who vote against it claim that the semantics of multiple instances can be tricky, so they would rather not allow it. For a parser, disallowing it would require more effort than allowing it. However, from the standpoint of tools using the parser, there might be some analyses that would need to distinguish different instances explicitly, which could result in more implementation effort. Still, the majority of the participants are in favor, so we recommend including this language feature.

Summary of the Findings from the Second Community Questionnaire

Although 16 participants are not the most accurate representation of the whole SPLE community, it still represents the part that is most committed to finding a common language. This is reflected in the result and the comments on whether participants would also agree to the other language, should the majority vote for that one (cf. Figure 6.1b). Multiple participants commented that

https://doi.org/10.24355/dbbs.084-202008211051-0

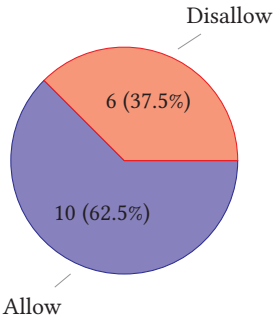


Figure 6.10.: Voting results on whether participants would allow multiple instantiation using the *as* keyword.

finding a common language is more important than finding the one with the best fit. With 14 of 16 participants voting for yes and only two voting for no, we are optimistic that the community will indeed agree to one language. Along with suggestions by the community and our own experience from implementing the parser library, we drop support for the *refer* keyword, in favor of simpler semantics and implementations.

6.1.3. Results of the Student Questionnaire

To determine which language concept is better for teaching and learning, we also sent the same questionnaire to students who are currently hearing an SPL course. As they are computer scientists who are in the process of learning the concepts of SPLE, they have a unique perspective on this aspect. We could only reach a few students, so we received only six replies to the questionnaire. Thus, we do not present detailed results of all the questions, as in the previous section, but focus on the main findings instead.

Overall the response is similar to that of the SPL community. Out of the six students, four prefer the nesting language concept over the referencing one. In turn, the responses for the nesting concept were mostly positive, while they were mixed for the referencing one. Comments suggest that participants find the referencing notation harder to read than the nesting one.

Regarding teaching and learning, five of six students agree that the nesting concept is well suited. In contrast, for the referencing concept, only two agree, while the other four students disagree or strongly disagree. Responses for the composition mechanism are mixed. One half agrees that it is well suited for teaching and learning, while the other half disagrees. However, all except one student indicated that they only work on small feature models with up to 100 features. In the course, students usually do not work on larger models, so a mechanism for composition and decomposition might not be important to them. Still, all the students are in favor of allowing the inclusion of multiple instances of the same model using different aliases. They value the possibilities for reuse that this feature enables.

The main outcome of this questionnaire is that students prefer the nesting concept and deem it well suited for teaching and learning. This is in line with the assessment of the community.

https://doi.org/10.24355/dbbs.084-202008211051-0

6.1.4. Summary of the Findings from the Second Questionnaire

We submitted a second questionnaire to both the SPL community and students who are currently enrolled in a course on SPLs. In both cases, the majority of the participants prefer the nesting language concept over the referencing one. The overall rating for it is positive and the more detailed criteria such as style, teaching and learning, or complexity are well regarded likewise. The referencing language concept received worse ratings in every aspect. Hence, we settle on the nesting concept to be our proposal for the Universal Variability Language.

From the feedback, we gather that the *refer* keyword with its semantics is considered too complex. We made the same observation while implementing the tool support. Thus, we drop the support for it to arrive at a simpler and more consistent language. We show the updated grammar of the final language in Listing A.1.

6.2. Evaluating Against the Requirements

After discussing the results of the second questionnaire in the previous section, we turn towards the collected requirements to evaluate how well the proposed language conforms to them. Since we settled on the nesting language concept to be the final language, from now on, we limit our discussions to that one. In the following sections, we revisit the general design guidelines (cf. Section 3.1) and the collected scenarios (cf. Section 3.2). Finally, we compare our language to the existing languages in Section 6.2.3.

6.2.1. Evaluating Against the Design Guidelines

In Section 3.1, we present and discuss general design guidelines for designing DSLs. The main recommendations Karsai et al. [KKP+14] give are to reuse, simplify, and to be consistent. In the following sections, we discuss in more detail if and to what extent our language adheres to those recommendations.

Reuse Karsai et al. [KKP+14] recommend reusing as much as possible from other languages. This includes both concrete and abstract syntax. The idea is to reduce the implementation effort, reuse good practices, and make the new language easy to learn.

Most concepts in UVL are reused from various programming or variability languages. For the abstract syntax, we reuse the general structure of feature diagrams. The model is a tree structure of features, which have groups as children, which in turn have other features as children. Additionally, there is a list of cross-tree constraints. This structure will be immediately familiar to anyone familiar with feature diagrams. The choice to use nesting and indentation to represent that structure, without requiring semicolons at the end of the line, is reused from Clafer (cf. Listing 3.5) or Python. To include other feature models, we incorporate namespaces. The semantics of namespaces are inspired by the package and import system used in Java, with individual segments of the namespace declaration describing the path to the file containing the model. The syntax of attributes is inspired by JSON objects and Clojure maps. Keywords that

https://doi.org/10.24355/dbbs.084-202008211051-0

are used in the constraints and consist only of a single symbol are inspired by C-like languages, which use the pipe symbol ('|') for an *or* and the ampersand symbol ('&') for an *and*.

Simplicity We try to keep the language as simple and obvious as possible. When starting with a simple model, the general structure from feature diagrams can be reused. Only when additional information needs to be specified, does the user have to learn the single concept of attributes. Since attributes are simply key-value pairs, they are a powerful tool with a simple syntax. If the model then becomes too large to be specified reasonably in a single file, there will be one more concept to learn, which is the composition mechanism. Since it is very similar to Java imports, its syntax and semantics are also familiar and easy to learn.

Consistency In feature diagrams, there is an inconsistency in where group membership is specified. While *or* and *alternative* groups are located at the parent, optional and mandatory features are specified at the children. We made this consistent in our language by introducing groups for optional and mandatory features, which can be used the same way as the other group types. However, one design decision might be called inconsistent: We allow specifying constraints in two places, although the constraints themselves still have the same syntax. In a separate list at the end of the file and within attributes. This way, users can choose the one that is better suited for their models. When the list of constraints is small, the separate list can give a good overview of all constraints. However, when the list becomes large enough to be incomprehensible, specifying constraints near involved children as an attribute might be more beneficial. We estimate that the benefits of having these two options for different use cases outweigh the downsides associated with this inconsistency.

6.2.2. Evaluating Against the Collected Scenarios

In addition to the general guidelines, we discussed and discerned scenarios and requirements by the SPL community in Section 3.2. In this section, we evaluate how well the language enables the selected scenarios and fulfills the resulting requirements.

Exchange The exchange scenario requires a textual language that is well documented, extensible, and provides a default library with a parser and a printer. In Section 4.2, we give an informal description of the language, followed by a formal grammar in Section 5.2. That serves as the documentation for the language. We describe the provided default library and its usage in Section 5.3. Extension of the language with tool-specific data is enabled through the attribute system. We demonstrate the feasibility by integrating the library as a further language into the FeatureIDE tool (cf. Section 5.4).

Mapping to Implementation The main requirement of this scenario is to mark whether specific features have a corresponding implementation artifact. This information can then be used in analyses to exclude features that are only meant for structuring, instead of introducing an actual feature to the product. This information can be specified in UVL with a boolean attribute,

https://doi.org/10.24355/dbbs.084-202008211051-0

which is usually called *abstract*. Furthermore, implementation artifacts can be referenced from other attributes, enabling traceability between specification and implementation or other artifacts, such as requirements or tests.

Teaching and Learning For this scenario, the language should be presentable with a few slides and reuse concepts that are familiar to computer science students, so they can learn the language quickly. As described in Section 6.2.1, we reuse many concepts from existing languages. The overall syntax is comparable to that of Python, which is familiar to many computer science students and the structure mostly adheres to that of feature diagrams. This way, computer scientists with rudimentary knowledge in the field of SPLs should be able to become familiar with the language quickly.

Storage This scenario describes the efficient storage, by the use of a textual syntax. We analyze the space efficiency when storing huge models in Section 6.3.3. The space efficiency of UVL is among the best of the compared textual formats.

Decomposition and Composition We provide a simple mechanism for composition and decomposition that is inspired by the package and import system that is incorporated in Java. Models can be included in another model as submodels using its namespace or an alias. We demonstrate that it can be easily integrated into tools by integrating it into FeatureIDE. Tools that do not support a similar composition mechanism can still use decomposed models by merging them into a single model and prepending each feature name with the alias it is imported with.

6.2.3. Comparison with Existing Languages

Each language author has reasoning behind their language design. That is one reason why Karsai et al. [KKP+14] recommend reusing language features from other languages. In this section, we put our language into the context of the existing ones. However, comparing multiple languages is a difficult task, as design decisions can rarely be considered in isolation. Thus, we focus on the categories we compared in Table 3.2 and a few additional noteworthy features.

UVL uses nesting to represent the hierarchy, indentation to represent blocks, line breaks to end a line, and full-length keywords, but as few as possible. With these design decisions, it would be similar to Clafer. However, with the constraint location as either a separate list or in-line and groups between the parent and its children, it introduces a unique combination of design decisions that is different from the other languages considered here. Also, Clafer in particular differs significantly in the abstract syntax from UVL, where there is no simple concept of a feature. Instead, a "Clafer", which entangles feature and class modeling, is used. It unifies notations, but introduces more complexity, reducing analyzability. We reuse the concept of different sections in the file from the VM approach (cf. Listing 3.10), indentation from Clafer or SXFM, and cardinalities from VSL. These design choices reflect the opinions of the majority of the community, which they expressed in the first questionnaire (cf. Section 3.4). With this combination, we hope that the community can agree to build upon it. We visualize

https://doi.org/10.24355/dbbs.084-202008211051-0

the similarities and differences from other languages in Table 6.1. For reference, we also include the referencing language concept in the table.

Language	Hierarchy	Blocks	Keywords	Line Endings	Constraint Location	Location of Groups
Concept 2	reference	indentation	minimal	new line	both	parent
FDL	reference	none	full	new line	separate	parent
GUIDSL	reference	none	symbols	semicolon	separate	parent[1]
FAMILIAR	reference	none	symbols	semicolon	separate	parent
PyFML	nesting	[]	full	semicolon	separate	parent
Clafer	nesting	indentation	minimal	new line	in-line	parent
UVL	nesting	indentation	minimal	new line	both	between
SXFM	nesting	indentation	symbols	new line	separate	between
VSL	nesting	()	minimal	semicolon	separate	between
TVL	nesting	{}	full	semicolon	in-line	between
μTVL	nesting	{}	full	semicolon	in-line	between
VELVET	nesting	{}	full	semicolon	in-line	between
VM	nesting	{}	minimal	new line	separate	between
IVML	nesting	{}	full	semicolon	in-line	n/a

[1] for "or" groups also the parent's parent

Table 6.1.: Summary of how the design decisions of UVL and the referencing language concept regarding the concrete syntax compare to presented languages. Decisions that are the same as in one of the concepts are accentuated with the corresponding color.

6.2.4. Summary of the Evaluation Against the Requirements

In the previous sections, we revisited the requirements for the language we determined in Chapter 3. We can justify each design decision against the general design guidelines from Section 3.1 and are confident that we made reasonable trade-offs, where necessary. We discussed how each of the selected scenarios from Section 3.2 can be enabled by our language and which design decisions lead to the fulfillment of the scenario's requirements. The language can be used to cover each relevant scenario. Finally, we put our language into context with the existing languages from Section 3.3. Our language features a unique combination of design decisions that is not yet present in other languages we are aware of. We pick and reuse good practices from a few of those languages, to arrive at a simple, yet sufficiently expressive and consistent language.

https://doi.org/10.24355/dbbs.084-202008211051-0

6.3. Evaluating Scalability

Lack of scalability is one of the main criticisms of the indentation-based format that came up in discussions with the community and comments of the questionnaires. In this section, we analyze what criteria for an approach that "scales" are and how well UVL fulfills them. For this, we consider two large real-world models (Section 6.3.1), explore which properties a text file must have to be considered "editable", and check how UVL representations of the example models fulfill these properties (Section 6.3.2). Finally, we analyze the space efficiency of different languages when storing these models in Section 6.3.3.

6.3.1. Introducing Real-World Models

Knüppel et al. [KTM+17] provide 127 large real-world feature models to be used for various analyses. For our analyses, we focus on the two largest ones. One is called Automotive02_V4, which is an obfuscated model from an industrial partner. It has a total of 18,616 features. The second model we consider is extracted from the Kconfig models of an older Linux kernel in version 2.6.33-rc3. This model has 6,467 features.

Both models would be impractical to handle in a single file. Thus, they are split into smaller submodels, which are then composed into the larger model. The Automotive model consists of 47 submodels, while the Linux kernel is decomposed into 675 submodels. We use these two models for the following analyses regarding editability and storage space efficiency.

The model of the Linux kernel is originally specified in Kconfig files. Knüppel et al. [KTM+17] translated those models to a single large feature model in the FeatureIDE format. To recover the former separation into submodels, we split the available model along the original Kconfig files. This step does not require a complete parser for the Kconfig format, as the overall structure has been retained in the feature model. This way, we only have to extract a list of feature names from each Kconfig file. We use those lists to separate the corresponding features from the feature model into their own submodels. For reference, we provide the code for this separation, the code for the following analyses, and the used models in a public repository on GitHub.[1]

6.3.2. Editing Large Models

In this section, we briefly analyze the characteristics of text editors to derive metrics to measure if and how well a textual file can be edited. We use these metrics to check how well the example models can be edited or, in other words, how well the UVL approach scales to large models.

Properties of an Editable Model

Different text editors provide vastly different editing features that can help with large files. Such features can include folding of subtrees, automatic matching of parentheses, editing at multiple caret positions simultaneously, or a powerful search-and-replace engine.

[1] `https://github.com/neominik/towards-uvl`

https://doi.org/10.24355/dbbs.084-202008211051-0

Assuming an editor without these supporting features, the simplest metric is whether the whole file fits on a single screen. If the model is small enough for that, every part of it can be seen at the same time, so keeping an overview and editing at any location is simple. Common computer screens with sizes between 13 and 24 inches diagonally (ca. 33 - 61 cm) can fit roughly 50 to 60 lines on a single page. However, this criterion is quite restrictive and probably unrealistic for real-world models.

A weaker, but more realistic, criterion would be to require only subtrees a certain level deep in the hierarchy to fit on a screen. This would imply that only a subsystem can fit on the screen, so it can still be edited with ease, while edits that concern features across subtrees would be more difficult. A related metric that is easier to measure is the maximum distance of two sibling features on a certain level. When the distance between two siblings is small enough for the screen, it implies that the subtree between the siblings also fits on the screen. An analogy can be made to recommendations about the sizes of classes and methods in software code. While a class might not fit on a single screen, a single method certainly should (Martin even recommends that functions should be no bigger than four lines [Mar08, p. 34]). What is the equivalence of a method in a feature model? A class consists of methods that describe one functionality. So in a feature model that consists of subtrees, there should be subtrees at various levels that can be considered analogous to methods and thus should fit on a single screen. Cross-tree constraints usually exist in their own list, separate from the context of the tree. Hence, to create or edit constraints, the involved features have to be known, requiring an overview of the feature tree. However, new constraints can be created at the end of the list and finding constraints to edit can be done by searching for them. This can be done even in large models, where the criterion regarding the size of the feature tree does no longer hold, so we do not consider constraints specifically for this analysis.

Another aspect of editability is the amount of indentation that occurs before the actual content. Assuming 150 characters can fit into a single line, then probably not more than half of that should consist of whitespace. We assume further a tab size of four spaces. That would allow for 18 tabs, before reaching the middle of the line. This criterion is probably the weakest for real-world models, as a model with such a deep nesting level would allow for far more features than could fit on the screen in most cases. However, this is not generally true for unbalanced trees, so we evaluate the criterion nonetheless.

Analyzing Real-World Models in UVL

Now that we defined different metrics to judge the editability of textual files, we can analyze the example models expressed in UVL. Without references, so all features in a single file, 18,616 or even 6,467 features would never fit on a screen. However, this is true for any notation. This is why these models are decomposed into submodels. 47 submodels for the Automotive model and 675 submodels for the Linux kernel.

In Figure 6.11, we show box plots of the number of features per submodel for both Automotive and Linux models. As these models do not use attributes, there is usually only one line per feature. With this, we can deduce that most models are small enough, so the entire feature tree

https://doi.org/10.24355/dbbs.084-202008211051-0

can fit on a screen. The median model size of Linux submodels is only three features, with the median for Automotive at 57.5. For Linux, 97% of the submodels have fewer than 60 features. For Automotive, this percentage is still at 49%. With this criterion alone, we can say that these real-world models are for the most part still easily editable.

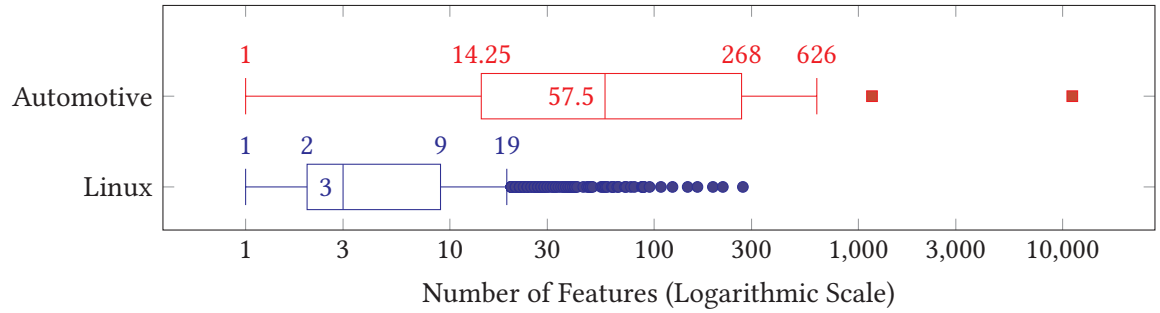


Figure 6.11.: Box plot of the number of features per submodel for Automotive and Linux on a logarithmic scale.

Considering the weaker criterion, the maximum distance between two siblings, the number of editable models increases significantly. In Figure 6.12, we show box plots of the maximum distance between siblings on the first level in the hierarchy. In the Automotive model, 77% of the submodels have a maximum sibling distance of fewer than 60 lines in the file, with the median at only eleven lines. When considering sibling distances on the second level and below, this distribution shifts further. Then, 87% of the models have a maximum sibling distance of fewer than 60 lines, so the vast majority of the models can still be edited. Linux submodels are smaller, so the maximum sibling distance is also much smaller for most models. 75% of the submodels have a maximum sibling distance of zero. This means that only leaf features have siblings in those models. Of the 675 submodels, 667 have a maximum sibling distance of less than 60. That is more than 98% of the models.

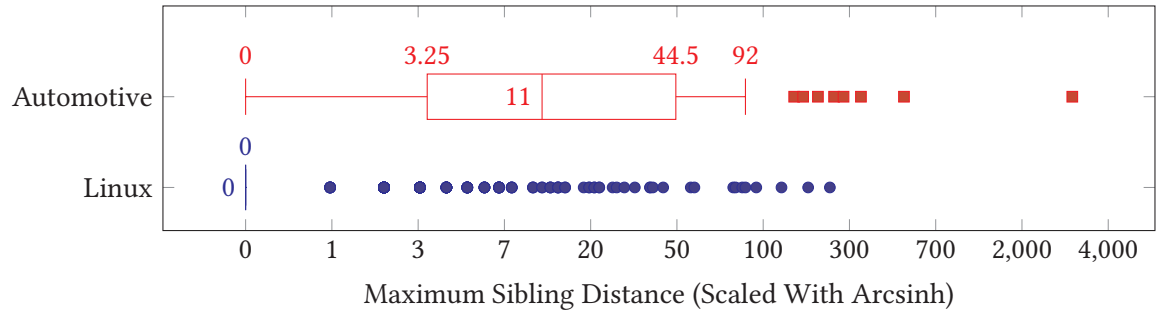


Figure 6.12.: Box plot of the maximum distance (in lines) between two siblings for Automotive and Linux models.

The final metric we examine is that of the maximum indentation level in the submodels. In Figure 6.13, we show box plots of this metric for the submodels. All submodels in both example models have fewer than 18 tabs as their maximum level of indentation. Most submodels in

https://doi.org/10.24355/dbbs.084-202008211051-0

Automotive have between 3 and 9 tabs as their maximum indentation level. Linux submodels are again smaller with typically between 1 and 7 tabs. Even for large models, huge amounts of indentation are uncommon, as the level of indentation only grows logarithmically with the size of the model when the tree is mostly balanced. With high branching factors (the Automotive model has an average branching factor of approx. 10.4, Linux approx. 6.8), these models can fit many features with reasonable indentation.

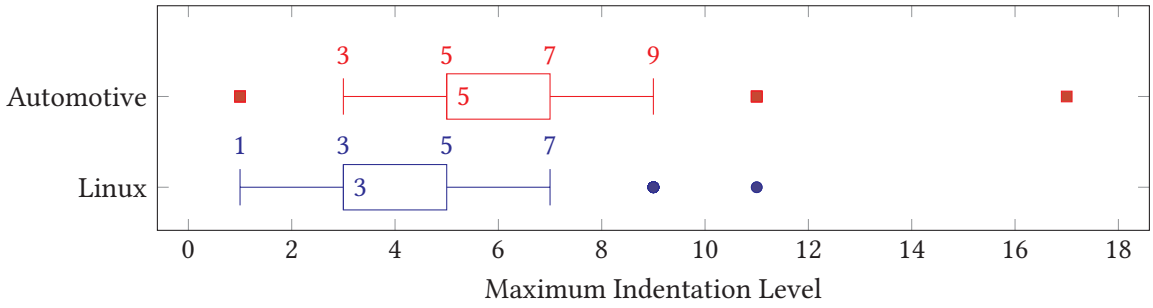


Figure 6.13.: Box plots of the maximum level of indentation (tabs) found in the submodels of Automotive and Linux.

We can conclude that even the largest real-world models can still be edited easily, except for a few outliers, because they are decomposed into submodels. Due to their size, these models already come decomposed. Otherwise, they would not be editable in any format. The majority of the submodels fit either on the screen entirely or have at least subtrees on the first or second level that do. Only a few outliers exist that would be more difficult to edit and could benefit from further decomposition. The maximum depth of indentation is small enough to not be a concern when editing these models.

We recommend that practitioners creating or maintaining large models should be guided by the design of the models considered here and decompose their models whenever they become too big. If possible, the number of features per model should be limited to 60. However, if that is infeasible, we recommend to at least create subtrees that are small enough to fit on a single screen.

6.3.3. Storage Efficiency

Another metric to consider when evaluating the data format is the file sizes it creates and how efficient the storage is. This consideration becomes less and less important with advancing storage capacities, faster Internet access, and advanced compression algorithms. However, it is still worth exploring briefly in response to the scenario about efficient storage (cf. Section 3.2).

To evaluate this metric, we use the same large real-world models from Section 6.3.1 and convert them into different formats. Initially, the models are available in the internal XML representation of FeatureIDE. FeatureIDE provides exporters for UVL, SXFM, and GUIDSL grammars. These formats already provide a good diversity in the design space of variability languages. SXFM uses nesting and indentation, similar to UVL, while the GUIDSL grammars

https://doi.org/10.24355/dbbs.084-202008211051-0

use references to represent the hierarchy. Thus, we use these formats for the comparison. In Figure 6.14, we give an overview of the space these formats require when storing the models. The results depend heavily on the individual models because different characteristics of a model result in different efficiencies in the formats. For instance, FeatureIDE stores constraints by its AST representation, pretty-printed with nesting and indentation. For few or simple constraints, this is not critical. However, for many complex constraints with deeply nested AST representations, the file size increases rapidly. This is especially pronounced for the Linux model, where the FeatureIDE format uses 90% of the file size for those constraints. The SXFM format is more sensitive to the length of the feature names, as the exporter writes every feature name twice. Once for the name and once for the unique ID that is required for each feature. The file size could be smaller if the exporter used shorter IDs, but incrementing integers would decrease the readability of constraints and meaningful short IDs would have to be provided by a human. This is the reason for the large file size of the Automotive model in SXFM, which has many features with long names, due to the obfuscation of those names.

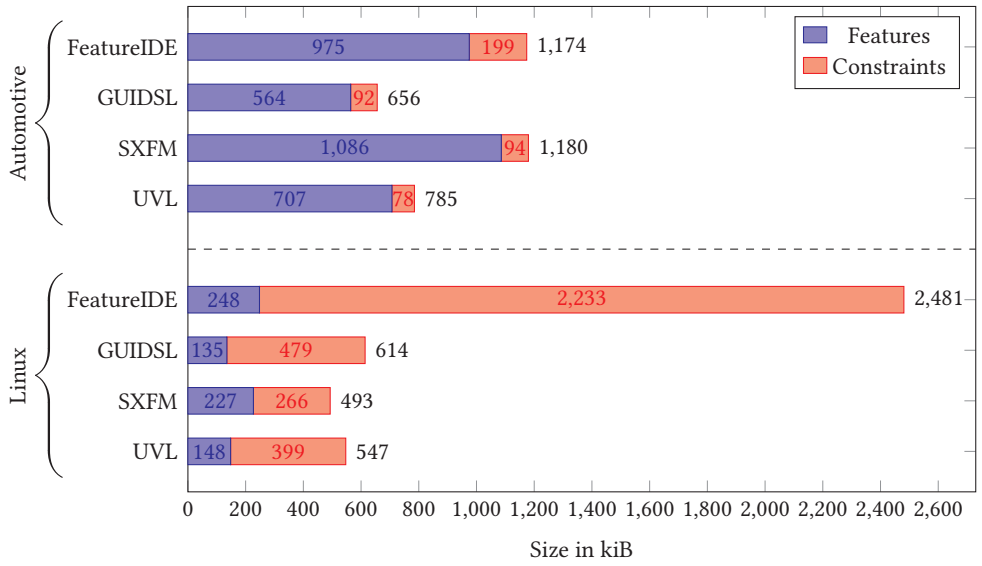


Figure 6.14.: Comparison of the file sizes for different formats. Split into the size for storing features and constraints.

The FeatureIDE format consistently produces one of the largest file sizes. For SXFM it depends heavily on the model. The Automotive model has its largest file size in SXFM, while for the Linux model it produces the smallest file. GUIDSL and UVL perform similarly and produce consistently small file sizes.

Another measurement to consider for efficient storage is the time it takes to store and retrieve models. While reading is very quick for all formats, finishing well under one second, the case is different for writing. FeatureIDE, GUIDSL, and UVL perform similar to reading, finishing also within well under a second for both models. The SXFM exporter, however, first converts all constraints into the conjunctive normal form to avoid constructs that are unsupported in SXFM, such as implications. This resulted in an export time of 30 minutes for the initial conversion

https://doi.org/10.24355/dbbs.084-202008211051-0

from the FeatureIDE representation of the Linux model to SXFM. The export was performed with FeatureIDE version 3.6.2 on an AMD Ryzen 5 3600. As we can see, supporting propositional logic in UVL is not only useful when modeling constraints, but also when storing them, avoiding expensive transformations.

6.4. Summary of the Evaluation

In this chapter, we evaluated the language concepts from many different angles. We conducted a second questionnaire, revisited the requirements from Chapter 3, and considered how well the language is suitable for large scale models.

The responses from the second questionnaire, which were collected from members of the SPL community and SPL students, are in favor of the nesting language concept. It is rated positively, both overall and in more detailed categories such as style, teaching and learning, and complexity. Since the referencing language concept is preferred by fewer participants and the feedback for that language is less positive, we settle on the nesting language concept to become the base language level of UVL.

We carefully consider the requirements from Chapter 3. These include general design guidelines, collected scenarios from the community, and a comparison to existing languages. The language meets the requirements and enables the scenarios. It makes reasonable trade-offs where necessary. Comparison to existing languages shows that it has similarities to many of the languages, but differs at least in a few aspects from each one.

Finally, we investigate how well the language is suitable for large real-world models. We use models with 18,616 and 6,467 features to evaluate how well they can be edited in a text editor and how space efficient they can be stored in UVL. These real-world models are composed of smaller submodels, many of which still fit on a single screen in UVL. For the majority, at least the subtrees beneath the first or second hierarchy level can fit on a screen. We recommend to keep models small and decompose them when necessary. Ideally, a single model has less than 60 features, although a weaker criterion of having subtrees on the first or second hierarchy level with less than 60 features might still be reasonable. Considering the space efficiency of the UVL format, we determine that it consistently produces good results, similar to the GUIDSL grammars.

For a definite assertion of the usability, a large-scale usability study comparing different modeling languages for specific tasks would have to be conducted. We consider this a task for future work when the community has agreed upon a language.

https://doi.org/10.24355/dbbs.084-202008211051-0

https://doi.org/10.24355/dbbs.084-202008211051-0

7. Related Work

In this chapter, we discuss publications related to textual variability languages. First, we present languages and approaches in chronological order of publication. Afterward, we look at efforts to arrive at a common or standardized variability language. Finally, we consider survey papers, comparing multiple textual notations.

The first attempt to arrive at a textual notation for the feature diagram proposed by Kang et al. [KCH+90] is the Feature Description Language (FDL) by van Deursen & Klint [vDK02] in 2002. The main goal of FDL is to formalize the notation of feature diagrams by providing a well-defined textual syntax. At the time, the tool support for feature diagrams was still lacking. Thus, van Deursen & Klint also made an effort to provide good tool support along with the language. Another early approach with similar goals are the GUIDSL grammars, published by Batory [Bat05] in 2005. Batory expresses feature diagrams using grammars and combines them with propositional formulas. This approach enables the use of SAT solvers to improve the existing tool support with analyses and validation of models and configurations.

Starting in 2009, many new languages and approaches were proposed in rapid succession. Mendonça et al. [MBC09] developed the Software Product Lines Online Tools (S.P.L.O.T.). It uses the Simple XML Feature Model (SXFM) as its storage format, which uses XML for the meta-data and an embedded DSL for the actual features. As part of the Compositional Variability Management Framework (CVM), Abele et al. [APS+10] introduce the Variability Specification Language (VSL). It is intended to evaluate research approaches that were developed in multiple industrial cooperations. At the same time, Boucher et al. [BCF+10] introduce the Text-based Variability Language (TVL). The motivation for TVL is to provide a language that supports large-scale models. To achieve that, it applies mechanisms for modularization of feature models. Additionally, TVL aims at offering a light and comprehensive syntax that can also be used for the storage of models. Later that year, Clarke et al. [CMP+10] present the Micro Text-based Variability Language (μTVL) as part of the ABS language. It is supposed to be a simplified version of TVL. By deliberately focusing on the essential parts of feature modeling, μTVL strives to simplify the manipulation of feature models. Clafer is proposed by Bąk et al. [BCW11]. Its main goal is to unify feature modeling and meta-modeling. This is achieved by combining features and classes into a single first-class concept. Furthermore, they seek to provide a concise concrete textual syntax. In the same year, Rosenmüller et al. [RST+11] suggest the VELVET language. They observed that in industrial projects, orthogonal aspects of feature models are often modeled independently to avoid scalability problems. The goal of VELVET is to provide a means to decompose models in the same way, while also offering to compose them again for analyses concerning the entire model. Additionally, configurations are supported. This way, there is a consistent language to cover both modeling and managing configurations. Inspired by Java, TVL, Clafer, OCL, UML, and others, Schmid et al. [SKE18] release the INDENICA Variability

https://doi.org/10.24355/dbbs.084-202008211051-0

Modeling Language (IVML) in its first version in 2012. Its main focus is to integrate product-line engineering and model-based development into a single approach. Another language, FAMILIAR, is released in 2013 by Acher et al. [ACL+13]. It also has the goal to enable large-scale models. For this, composition and decomposition, reasoning facilities, scripting capabilities, and modularization mechanisms are provided.

After a brief pause of a few years, PyFML by Azzawi [Azz18] appears in 2018. It aims at being more scalable than feature diagrams and at providing enough expressiveness for dealing with complex SPLs. The most recent attempt at a textual language, we are aware of, is called VM (Alférez et al. [AAG+19]). In this industrial project, a variability modeling approach specific to the video domain is developed. The domain's unique requirements made it necessary to consider various existing approaches and integrate them into a single tool. Although it was developed specifically for the video domain, its textual language can be used in other domains.

The languages presented so far all have their own specific goals and reasoning or are part of their own tool suite. However, none of them gained wide adoption across most variability tools. With our proposal, we try to provide a starting point for a language that could fill that spot. We argue that collecting feedback from the community during the development of a language is a vital component in this process. A previous effort to arrive at a standardized language was the Common Variability Language (CVL) [HMO+08; HWC12]. It was specified using the Meta-Object Facility (MOF) and was to be hosted at the Object Management Group (OMG) alongside their other standards. With CVL, one could add variability to other DSLs, providing means for wide adoption across domains. However, it failed due to legal reasons [BC19]. Also, the proposed visual notation was lacking in usability as found by Echeverría et al. [EFC+15]. The CVL documentation is still available as a revised submission to the OMG [Hau12]. Lately, a new effort has been started to arrive at a common language in September 2018 [BC19]. As described in Section 3.2, in the current stage, scenarios and requirements have been collected and ranked by the community. As part of this effort, Thüm et al. [TSS19] make suggestions regarding the expressiveness of a new language. When expressiveness is too high, adoption will be hindered by the effort to implement and integrate all the features into tools. In contrast, an expressiveness that is too low does not allow expressing real-world models from various domains. As a solution, Thüm et al. suggest resorting to different language levels. They could coincide with the expressiveness of different classes of solvers. We also acknowledge this expressiveness trade-off and propose a first, basic level, trying to strike a reasonable balance between complexity and usefulness.

Eichelberger & Schmid [ES13] compare ten different textual languages. They compare capabilities, constraints, configuration support, scalability, and additional language characteristics. This analysis has been updated and refined two times ([ES15; tBSE19]), considering new languages or new developments in already covered ones. Eichelberger & Schmid focus mainly on the exact feature set and expressiveness of those languages. We compare almost the same set of languages in our analysis. However, we focus more on the language's concrete textual syntaxes, by representing an example model in each of the different languages. This aspect has, to our knowledge, not yet explicitly been studied by other surveys.

https://doi.org/10.24355/dbbs.084-202008211051-0

8. Conclusion and Future Work

Numerous variability languages and approaches exist in the SPL community. Each one comes with a slightly different focus, goal, and set of language features. Thus, none of them enjoys wide support across tools, hindering effective communication and the exchange of models. However, a standard data format has many advantages for the community as a whole, including ease of collaboration, teaching and learning, benchmarking, and reusing analyses from other tools. In this thesis, we designed a proposal for a language that can be used as a basis to commit to such a common format. We incorporated extensive feedback from the community, both before and during the design, to increase the acceptance of the final language.

We derived and discerned the relevant requirements by considering four different kinds of sources. These sources are general guidelines on designing DSLs, scenarios that have been formulated and ranked by the SPL community, existing textual variability languages, and a questionnaire about the needs and preferences of the individual members of the community. While some decisions were clear from the requirements, others, such as the question of whether to use nesting or references to represent the hierarchy, resulted in a tie in the submissions to the questionnaire. With valid arguments for both approaches, we decided to create two proposals for a new language, to be examined side by side by the community. Key guiding principles while designing both languages were simplicity, familiarity, and flexibility. This way, we aim for the language to be easy to integrate into tools, natural to learn and teach, and able to express existing and future models from various approaches.

In a second questionnaire, we presented the two language proposals to receive feedback from the community and determine the preferred language. With both concrete proposals side by side, the questionnaire resulted in a clear choice. The community prefers the language concept that uses nesting and indentation to represent the hierarchy, which we call the Universal Variability Language (UVL). That language is well received, with good ratings overall. Participants can represent their models in the language and think that it is well suited for teaching and learning. Reflection on the requirements shows that the language enables the relevant scenarios and conforms to general best practices and guidelines. Furthermore, it is well suited for scalability. The main factor for this is not the syntax itself, but the composition mechanism. Large-scale models, such as the Linux kernel, which are decomposed into smaller submodels already, can be represented with the same decomposition in UVL. This way, most models are sufficiently small to be edited comfortably with text editors, even though we expect the majority of the models to be created as exports from tools.

To enable rapid integration into most of these tools, we provide a small default implementation that can be used as a Java library. It includes a parser and a printer. We demonstrated the

https://doi.org/10.24355/dbbs.084-202008211051-0

library's applicability by including it as an additional format into the FeatureIDE tool. For tools written in other languages than Java, we provide a grammar that can be used to generate appropriate implementations.

The next step for the community is to commit to a common language. They may use our language proposal and the results of the questionnaires to guide their decision. We urge practitioners who are developing variability tools to then integrate support for that language in their tools. Only with widespread tool support can a language gain adoption.

Additionally, syntax support for textual editors could be developed to ease the direct editing of models by hand (e.g., when teaching it to students). Ideally, this editing support is not limited to a single tool but can be used in all major editors and IDEs. To achieve this, the Language Server Protocol (LSP) could be used.

We focused mainly on the concrete syntax and the scope of the language, as these are the most critical points for acceptance. For the future, it would be beneficial to define a formal semantics for the language. This way, misunderstandings on how certain language constructs have to be interpreted can be minimized. However, the existing intuitive understanding of how to translate the language concepts to feature diagrams might be precise enough in the beginning.

For this base language level, we traded expressiveness for simplicity and analyzability with SAT solvers. Though it is extensible with attributes, these cannot be used in constraints and constraints themselves are limited to propositional logic. For more involved use cases, additional language levels extending the first one (e.g., by providing more powerful constraints) could be defined. With these levels, the language could apply to a wider range of models at the cost of increased complexity and decreased analyzability.

Finally, while this language focuses on specifying the variability in models, the configuration step could also benefit from a common format. When the community commits to a variability language, a configuration language may be defined. It could build on the same principles and results of this language, employing a similar syntax and enabling exchange between tools.

https://doi.org/10.24355/dbbs.084-202008211051-0

Bibliography

[AAG+19] Mauricio Alférez, Mathieu Acher, José A. Galindo, Benoit Baudry, and David Benavides. Modeling variability in the video domain: language and experience report. *Software Quality Journal*, 27(1):307–347, 2019. DOI: 10.1007/s11219-017-9400-8 (Cited on pages 25, 78).

[ABK+13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, Berlin, Heidelberg, 2013 (Cited on pages 3, 4).

[ACL+11] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Slicing Feature Models. In *IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE)*, pages 424–427, Washington, DC, USA. IEEE, 2011. ISBN: 978-1-4577-1638-6. DOI: 10.1109/ASE.2011.6100089 (Cited on page 45).

[ACL+13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. FAMILIAR: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)*, 78(6):657–681, 2013. DOI: 10.1016/j.scico.2012.12.004 (Cited on pages 16, 21, 78).

[ANT20a] ANTLR. Grammars written for ANTLR v4. GitHub repository, 2020. Available online at https://github.com/antlr/grammars-v4; visited on June 11th, 2020. (Cited on page 50).

[ANT20b] ANTLR / Terence Parr. ANTLR. Website, 2020. Available online at https://www.antlr.org; visited on June 11th, 2020. (Cited on page 50).

[APS+10] Andreas Abele, Yiannis Papadopoulos, David Servat, Martin Törngren, and Matthias Weber. The CVM Framework - A Prototype Tool for Compositional Variability Management. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 101–105, January 2010 (Cited on pages 23, 77).

[Azz18] A.F. Al Azzawi. PYFML - A Textual Language for Feature Modeling. *Int'l Journal of Software Engineering & Applications*, 9(1):41–53, January 2018. ISSN: 0975-9018. DOI: 10.5121/ijsea.2018.9104 (Cited on pages 21, 78).

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 7–20, Berlin, Heidelberg. Springer, 2005 (Cited on pages 1, 20, 49, 77).

[BC19] Thorsten Berger and Philippe Collet. Usage Scenarios for a Common Feature Modeling Language. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, volume B of *SPLC '19*, pages 174–181, Paris, France. Association for Computing Machinery, 2019. DOI: 10.1145/3307630.3342403 (Cited on pages 2, 13, 14, 16, 18, 37, 78).

https://doi.org/10.24355/dbbs.084-202008211051-0

[BCF+10] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing TVL, a Text-based Feature Modelling Language. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 159–162, January 2010 (Cited on page 77).

[BCW11] Kacper Bąk, Krzysztof Czarnecki, and Andrzej Wąsowski. Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled. In *Software Language Engineering*, pages 102–122, Berlin, Heidelberg. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-19440-5_7 (Cited on pages 22, 77).

[BEH+17] Meinte Boersma, Sebastian Erdweg, Angelo Hulshout, Steven Kelly, Tijs van der Storm, and Markus Völter. Language Workbench Challenge — Comparing Tools of the Trade. Website, 2017. Archived version available online at http://web.archive.org/web/20170628212322/http://www.languageworkbenches.net/; visited on June 11th, 2020. (Cited on page 50).

[Big20] Big Lever Software Inc. Gears: A Software Product Line Engineering Tool. Website, 2020. Available online at https://www.biglever.com/solution/product.html; visited on February 26th, 2020. (Cited on page 1).

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 7:1–7:8, Pisa, Italy. ACM, 2013. ISBN: 978-1-4503-1541-8. DOI: 10.1145/2430502.2430513 (Cited on pages 5, 54).

[BTS19] Paul Maximilian Bittner, Thomas Thüm, and Ina Schaefer. SAT Encodings of the At-Most-k Constraint: A Case Study on Configuring University Courses. In *Software Engineering and Formal Methods*, pages 127–144, Oslo, Norway. Springer, September 2019. DOI: 10.1007/978-3-030-30446-1_7 (Cited on page 46).

[Bün19] Hendrik Bünder. Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages. In *Proc. of the 7th Int'l Conf. on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 131–142, 2019 (Cited on page 49).

[CBH11] Andreas Classen, Quentin Boucher, and Patrick Heymans. A Text-Based Approach to Feature Modelling: Syntax and Semantics of TVL. *Science of Computer Programming (SCP)*, 76(12):1130–1143, 2011. Special Issue on Software Evolution, Adaptability and Variability (Cited on pages 16, 24).

[Cen14] Centrum Wiskunde & Informatica. Rascal MPL. Website, 2014. Available online at https://www.rascal-mpl.org; visited on June 11th, 2020. (Cited on page 51).

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing Cardinality-Based Feature Models and Their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005. DOI: 10.1002/spip.213 (Cited on page 30).

https://doi.org/10.24355/dbbs.084-202008211051-0

[CMP+10] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability Modelling in the ABS Language. In *Formal Methods for Components and Objects*, volume 6957, pages 204–224, November 2010. DOI: 10.1007/978-3-642-25271-6_11 (Cited on pages 24, 77).

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2001. ISBN: 0-201-70332-7 (Cited on pages 1, 3, 4).

[CPB+07] Daniela Cruz, Maria João Pereira, Mario Berón, Rúben Fonseca, and Pedro Henriques. Comparing generators for language-based tools. In *Proc. of the 1st Conf. on Compiler, Related Technologies and Applications*, pages 27–50. Universidade da Beira Interior, 2007 (Cited on page 50).

[DRG+07] Deepak Dhungana, Rick Rabiser, Paul Grünbacher, Klaus Lehner, and Christian Federspiel. DOPLER: An Adaptable Tool Suite for Product Line Engineering. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 151–152. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 2007. ISBN: 978-4-7649-0342-5 (Cited on page 49).

[Ecl20] Eclipse Fondation. Xtext - Language Engineering Made Easy! Website, 2020. Available online at https://www.eclipse.org/Xtext/; visited on March 12th, 2020. (Cited on pages 11, 51).

[EFC+15] Jorge Echeverría, Jaime Font, Carlos Cetina, and Oscar Pastor. Usability Evaluation of Variability Modeling by means of Common Variability Language. In *Proc. of the CAiSE 2015 Forum at the 27th Int'l Conf. on Advanced Information Systems Engineering*, pages 105–112, Stockholm, Sweden, June 2015. URL: http://ceur-ws.org/Vol-1367/paper-14 (Cited on page 78).

[EKR+11] Sebastian Erdweg, Lennart C.L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. SugarJ: Library-Based Language Extensibility. In *Proc. of the ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 187–188, Portland, Oregon, USA. Association for Computing Machinery, 2011. ISBN: 9781450309424. DOI: 10.1145/2048147.2048199 (Cited on page 50).

[ES13] Holger Eichelberger and Klaus Schmid. A Systematic Analysis of Textual Variability Modeling Languages. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, SPLC '13, pages 12–21, Tokyo, Japan. Association for Computing Machinery, 2013. ISBN: 9781450319683. DOI: 10.1145/2491627.2491652 (Cited on page 78).

[ES15] Holger Eichelberger and Klaus Schmid. Mapping the Design Space of Textual Variability Modeling Languages: A Refined Analysis. *Int'l J. Software Tools for Technology Transfer (STTT)*, 17(5):559–584, 2015. ISSN: 1433-2779. DOI: 10.1007/s10009-014-0362-x (Cited on page 78).

https://doi.org/10.24355/dbbs.084-202008211051-0

[EvdSV+15] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015. ISSN: 1477-8424. DOI: 10.1016/j.cl.2015.08.007. Special issue on the 6th and 7th Int'l Conf. on Software Language Engineering (SLE 2013 and SLE 2014) (Cited on page 50).

[FHB+14] Alexander Felfernig, Lothar Hotz, Claire Bagley, and Juha Tiihonen. *Knowledge-Based Configuration - From Research to Business Cases*. Morgan Kaufmann, Boston, 2014. ISBN: 978-0-12-415817-7. DOI: 10.1016/B978-0-12-415817-7.00001-3 (Cited on page 2).

[GK99] Andreas Günter and Christian Kühn. Knowledge-Based Configuration- Survey and Future Directions. In *XPS-99: Knowledge-Based Systems. Survey and Future Directions*, pages 47–66, Berlin, Heidelberg. Springer Berlin Heidelberg, 1999. ISBN: 978-3-540-49149-1 (Cited on page 2).

[Hau12] Øystein Haugen. Common variability language (CVL) - OMG® revised submission, 2012. OMG document ad/2012-08-05 (Cited on page 78).

[HJK+09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In *Model Driven Architecture - Foundations and Applications*, pages 114–129, Berlin, Heidelberg. Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-02674-4 (Cited on page 11).

[HMO+08] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 139–148, 2008 (Cited on page 78).

[HWC12] Øystein Haugen, Andrzej Wąsowski, and Krzysztof Czarnecki. CVL: Common Variability Language. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, SPLC '12, pages 266–267, Salvador, Brazil. Association for Computing Machinery, 2012. ISBN: 9781450310956. DOI: 10.1145/2364412.2364462 (Cited on pages 1, 78).

[Jav18] JavaCC Community. JavaCC — The most popular parser generator for use with Java applications. Website, 2018. Available online at https://javacc.org/; visited on June 11th, 2020. (Cited on page 50).

[Jet20] JetBrains s.r.o. MPS: The Domain-Specific Language Creator by JetBrains. Website, 2020. Available online at https://www.jetbrains.com/mps/; visited on March 11th, 2020. (Cited on page 11).

https://doi.org/10.24355/dbbs.084-202008211051-0

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report CMU/SEI-90-TR-21, Software Engineering Institute, 1990 (Cited on pages 4, 5, 77).

[KKP+14] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages, 2014. arXiv: 1409.2378 [cs.SE] (Cited on pages 13, 14, 66, 68).

[KTM+17] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is There a Mismatch Between Real-World Feature Models and Product-Line Research? In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 291–302. ACM, 2017 (Cited on pages 35, 40, 46, 70).

[KVR+20] Bernd Kolb, Markus Völter, Daniel Ratiu, Domenik Pavletic, Kolja Dummann, and Tamás Szabó. mbeddr - engineering the future of embedded software. Website, 2020. Available online at http://mbeddr.com/team.html; visited on June 11th, 2020. (Cited on page 49).

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, United States, 1st edition, 2008. ISBN: 978-0-13-235088-4 (Cited on page 71).

[Mar19] Mark Engelberg. Engelberg/instaparse. GitHub repository, 2019. Available online at https://github.com/Engelberg/instaparse; visited on June 11th, 2020. (Cited on page 50).

[MBC09] Marcílio Mendonça, Moises Branco, and Donald Cowan. S.P.L.O.T. - Software Product Lines Online Tools. In *Proc. of the ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 761–762, October 2009. DOI: 10.1145/1639950.1640002 (Cited on pages 23, 77).

[Mer10] Bernhard Merkle. Textual Modeling Tools: Overview and Comparison of Language Workbenches. In *Proc. of the ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 139–148, Reno/Tahoe, Nevada, USA. Association for Computing Machinery, 2010. ISBN: 9781450302401. DOI: 10.1145/1869542.1869564 (Cited on page 50).

[Met20] MetaBorg. The Spoofax Language Workbench. Website, 2020. Available online at https://metaborg.org/; visited on March 12th, 2020. (Cited on pages 11, 51).

[MHS05] Marjan Mernik, Jan Heering, and Anthony Sloane. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.*, 37:316–, December 2005. DOI: 10.1145/1118890.1118892 (Cited on pages 6, 7).

[Mic20] Microsoft Corporation. Official page for Language Server Protocol. Website, 2020. Available online at https://microsoft.github.io/language-server-protocol/; visited on June 11th, 2020. (Cited on page 49).

https://doi.org/10.24355/dbbs.084-202008211051-0

[MLW18] Hanspeter Mössenböck, Markus Löberbauer, and Albrecht Wöß. The Compiler Generator Coco/R. Website, 2018. Available online at http://ssw.jku.at/Coco/; visited on June 11th, 2020. (Cited on page 50).

[MTS+17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, Berlin, Heidelberg, 2017. ISBN: 978-3-319-61442-7. DOI: 10.1007/978-3-319-61443-4 (Cited on pages 1, 49).

[Obj17] Object Management Group®. OMG® Unified Modeling Language® (OMG UML®) Version 2.5.1. Website, 2017. Available online at https://www.omg.org/spec/UML/2.5.1/; visited on Mai 1st, 2020. (Cited on page 40).

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin, Heidelberg, November 2005 (Cited on pages 1, 3).

[Pet95] Marian Petre. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Commun. ACM*, 38:33–44, June 1995. DOI: 10.1145/203241.203251 (Cited on page 9).

[PHF14] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *Proc. of the ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 579–598, Portland, Oregon, USA. Association for Computing Machinery, 2014. ISBN: 9781450325851. DOI: 10.1145/2660193.2660202 (Cited on page 50).

[PP08] Michael Pfeiffer and Josef Pichler. A comparison of tool support for textual domain-specific languages. In *Proc. of the 8th OOPSLA Workshop on Domain-Specific Modeling*, pages 1–7, 2008 (Cited on page 50).

[pur20] pure::systems. pure::variants. Website, 2020. Available online at https://www.pure-systems.com/products/pure-variants-9.html; visited on February 26th, 2020. (Cited on pages 1, 49).

[RST+11] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Multi-Dimensional Variability Modeling. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 11–22, Namur, Belgium. ACM, January 2011 (Cited on pages 25, 77).

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *Proc. Int'l Conf. on Requirements Engineering (RE)*, pages 136–145, Washington, DC, USA. IEEE, 2006. ISBN: 0-7695-2555-5. DOI: 10.1109/RE.2006.23 (Cited on page 5).

[SKE18] Klaus Schmid, Christian Kröher, and Sascha El-Sharkawy. Variability Modeling with the Integrated Variability Modeling Language (IVML) and EASy-Producer. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, SPLC '18, page 306,

https://doi.org/10.24355/dbbs.084-202008211051-0

Gothenburg, Sweden. Association for Computing Machinery, 2018. ISBN: 9781450364645. DOI: 10.1145/3233027.3233057 (Cited on pages 26, 77).

[SKT+16] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 667–678, Austin, Texas. ACM, May 2016. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884823 (Cited on pages 30, 35, 40, 45).

[Stu97] Markus Stumptner. An Overview of Knowledge-Based Configuration. *AI Commun.*, 10(2):111–125, April 1997. ISSN: 0921-7126 (Cited on page 2).

[TAK+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45, June 2014. ISSN: 0360-0300. DOI: 10.1145/2580950 (Cited on page 1).

[tBSE19] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. Textual Variability Modeling Languages: An Overview and Considerations. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, SPLC '19, pages 151–157, Paris, France. Association for Computing Machinery, 2019. ISBN: 9781450366687. DOI: 10.1145/3307630.3342398 (Cited on pages 26, 78).

[TSS19] Thomas Thüm, Christoph Seidl, and Ina Schaefer. On Language Levels for Feature Modeling Notations. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, SPLC '19, pages 158–161, Paris, France. Association for Computing Machinery, 2019. ISBN: 9781450366687. DOI: 10.1145/3307630.3342404 (Cited on pages 19, 40, 78).

[VBD+13] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. 2013. ISBN: 978-1-4812-1858-0. URL: https://www.dslbook.org (Cited on pages 6–9).

[vDK02] Arie van Deursen and Paul Klint. Domain-Specific Language Design Requires Feature Descriptions. *Computing and Information Technology*, 10(1):1–17, 2002. DOI: 10.2498/cit.2002.01.01 (Cited on pages 20, 77).

[WSH13] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler Design: Syntactic and semantic analysis*. Springer, Berlin, November 2013 (Cited on pages 9, 10).

[Zip20] Roman Zippel. KConfig Documentation. Website, 2020. Available online at https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt; visited on February 26th, 2020. (Cited on page 1).

https://doi.org/10.24355/dbbs.084-202008211051-0

https://doi.org/10.24355/dbbs.084-202008211051-0

A. Appendix

A.1. First Questionnaire

For reference, we list the questions from the first questionnaire here.

Keyword length

1. What kinds of keywords do you prefer? (long, abbreviated, symbols)
2. Why did you choose the previous answer(s)?
3. Why not one of the other answers?
4. If you chose symbols before, what kinds of symbols do you prefer for different language constructs?
5. Other comments?

Line Breaks

6. How do you prefer to end your lines? (semicolon, line break, other…)
7. Why did you choose the previous answer(s)?
8. Why not one of the other answers?
9. Other comments?

Structuring

10. How do you prefer to add structure to files? (indentation, curly braces, parentheses, other…)
11. Why did you choose the previous answer(s)?
12. Why not one of the other answers?
13. Other comments?

Hierarchy

14. How do you prefer to specify hierarchy? (Embedded by nesting blocks beneath parents, Reference children from parents without nesting, other…)
15. Why did you choose the previous answer(s)?
16. Why not one of the other answers?
17. Other comments?

https://doi.org/10.24355/dbbs.084-202008211051-0

Location of Groups

18. Where do you prefer to specify group membership? (At the parent, On the children, In-between, other…)
19. Why did you choose the previous answer(s)?
20. Why not one of the other answers?
21. Other comments?

Groups vs. Cardinality vs. Constraints

22. How much power do you need? What is most intuitive? (Groups, Cardinality, Constraints on number of children, Constraints at least / at most on sets of features, other…)
23. Why did you choose the previous answer(s)?
24. Why not one of the other answers?
25. Other comments?

Scope

26. Which language features should be supported?
    - Default selections for configurations
    - Abstract features
    - Save entire configurations
    - Attributes for features
    - Reference other feature models
    - Feature model interfaces
    - Extension mechanism for arbitrary additional data (e.g., layout information)
27. Why did you choose the previous answer(s)? What are your use-cases for the selected features?
28. Why not one of the other answers?
29. Other comments?

Constraints

30. How much expressive power do you need for constraints? (Propositional logic (e.g., $A \implies B$), First-order logic (e.g., $\forall a \exists b(\text{IsLeaf}(a) \implies \text{Siblings}(a, b))$), other…)
31. Why did you choose the previous answer(s)?
32. Why not one of the other answers?
33. Other comments?

https://doi.org/10.24355/dbbs.084-202008211051-0

Separation of Concerns

34. Which information should be in-line instead of by reference in separate files/segments? Mark the ones that should be in-line. (Hierarchy, Abstract feature?, Constraints, Default selections, Configurations, Arbitrary additional data)
35. Why did you choose the previous answer(s)? How do you draw the line?
36. Why not one of the other answers?
37. Other comments?

Use or adapt existing serialization format?

Examples: XML, YAML, JSON, EDN, OpenDDL, …
Main advantage: Parsers for these formats already exist for many languages, which is a huge benefit.
Drawbacks: As they are made for generic data, they lack support for special constructs like constraints, making them much more verbose than a specialised DSL.

38. Are you in favour of using an existing format? (yes, no, undecided)
39. Why did you choose the previous answer? If yes, which format would you use?
40. What about the other side? What is the most important argument against it?
41. Other comments?

IDE support

Today we are used to the various comfort features modern IDEs offer above a plain text-editor. Stil opting to integrate the language into an existing IDE, probably by using a language workbench results in a considerable amount of tool lock-in.

42. How much tool lock-in are you prepared to endure?
    - None, provide an EBNF-spect only for maximum portability
    - A small default library and parser without external dependencies is fine
    - Editor integration for a specific IDE/Framework
    - Use a projectional editor to enable editing in different views, but make editing of the source basically impossible
    - I don't care, as long as I can port it for my tool
43. Why did you choose the previous answer(s)? If tool lock-in, which tool / IDE / Framework would you prefer?
44. Why not one of the other answers?
45. Other comments?

https://doi.org/10.24355/dbbs.084-202008211051-0

A.2. Second Questionnaire

For reference, we list the questions from the second community questionnaire here.

Introducing the Server Example

The graphical representation of feature models is quite consistent between tools. Textual representations, however, are not. This example will be the basis for showing the concrete syntax of the concepts for a textual language. Consider the following example of a configurable server, expressed as a feature model: (Figure 2.2)

1. Seeing that this example model is very small so it will easily fit on a screen in any representation, what is the size of the largest feature model you have worked with?
2. What is the size of Typical feature models you work with?

Language Concept 1: Nested Hierarchy

Here, we give a short introduction to the nesting language concept, as in Section 4.2.

3. How do you like this language concept?
4. Please indicate your level of agreement to the following statements.
   - The style is good
   - It is well suited for teaching and learning
   - It is too complex
   - It can easily be integrated into my tool
   - I can represent my models in this language
5. What would you change about it? Please elaborate.
6. How happy would you be with this language concept when your changes were applied?

Language Concept 2: Referenced Hierarchy

Here, we give a short introduction to the referencing language concept, as in Section 4.3.

7. How do you like this language concept?
8. Please indicate your level of agreement to the following statements. (same statements as above)
9. What would you change about it? Please elaborate.
10. How happy would you be with this language concept when your changes were applied?

https://doi.org/10.24355/dbbs.084-202008211051-0

Composition Mechanism

Here, we give a short introduction to the composition mechanism, as in Section 4.4.

11. How do you like this concept?
12. Please indicate your level of agreement to the following statements. (same statements as above)
13. What would you change about it? Please elaborate.
14. How happy would you be with this concept when your changes were applied?
15. Using namespace aliasing with ”as”, the same feature model could be included multiple times under different names in the same file. For instance, a car could have multiple seats with different configurations. Would you consider this possibility useful or rather disallow it?
16. Why did you choose the previous answer? Please elaborate.

Option Choice

Now that we presented the whole picture for both language concepts (including the composition mechanism), please indicate which of the language concepts you prefer.

17. If you had to choose one of the two presented language concepts, which one would it be?
18. Why did you chose the previous answer?
19. What bothers you about your previous answer?
20. In the event that most members of the community vote for the other language concept, would you still agree to that one?
21. In case you voted for ”no”, what would need to change for it to gain your acceptance?

https://doi.org/10.24355/dbbs.084-202008211051-0

```
FeatureModel = Ns? Imports? Features? Constraints?

Ns = <'namespace'> REF
Imports = <'imports'> (<indent> Import+ <dedent>)?
Import = REF (<'as'> ID)?

Features = <'features'> Children?
<Children> = <indent> FeatureSpec+ <dedent>
FeatureSpec = REF Attributes? Groups?
Attributes = (<'{'> <'}'>) | (<'{'> Attribute (<','> Attribute)* <'}'>)
Attribute = Key Value?
Key = ID
Value = Boolean | Number | String | Attributes | Vector | Constraint
Boolean = 'true' | 'false'
Number = #'[+-]?(0|[1-9]\d*)(\.\d*)?([eE][+-]?\d+)?'
String = #'"(?:[^"\\\n]|\\.)*"'
Vector = <'['> (Value <','?>)* <']'>
Groups = <indent> Group* <dedent>
Group = ('or' | 'alternative' | 'mandatory' | 'optional' | Cardinality)
        Children?
Cardinality = <'['> (int <'..'>)? (int|'*') <']'>

Constraints = <'constraints'> (<indent> Constraint+ <dedent>)?
<Constraint> = disj-impl | Equiv
Equiv = Constraint <'<=>'> disj-impl
<disj-impl> = disj | Impl
Impl = disj-impl <'=>'> disj
<disj> = conj | Or
Or = disj <'|'> conj
<conj> = term-not | And
And = conj <'&'> term-not
<term-not> = term | Not
Not = <'!'> term
<term> = REF | <'('> Constraint <')'>

indent = '_INDENT_'
dedent = '_DEDENT_'
<strictID> = #'(?!alternative|or|features|constraints|true|false|as
               |refer)[a-zA-Z][a-zA-Z_0-9]*'
<ID> = #'(?!true|false)[a-zA-Z][a-zA-Z_0-9]*'
REF = (ID <'.'>)* strictID
<int> = #'0|[1-9]\d*'
```

Listing A.1: Final grammar for the Universal Variability Language (UVL) in EBNF-like notation. All whitespace characters should be ignored automatically.

https://doi.org/10.24355/dbbs.084-202008211051-0