UVL: Feature modelling with the universal variability language

David Benavides, Chico Sundermann, Kevin Feichtinger, José A. Galindo, Rick Rabiser, Thomas Thüm

PII: S0164-1212(24)00370-4
DOI: https://doi.org/10.1016/j.jss.2024.112326
Reference: JSS 112326

To appear in: *The Journal of Systems & Software*

Received date : 15 March 2024
Revised date : 1 October 2024
Accepted date : 18 December 2024

UVL: Feature Modelling with the Universal Variability Language

David Benavides[a], Chico Sundermann[b], Kevin Feichtinger[c], José A. Galindo[a], Rick Rabiser[d], Thomas Thüm[e]

[a]*Department of Computer Languages and Systems, I3US, Universidad de Sevilla, Av. Reina Mercedes, Seville, 41012, Spain {benavides, jagalindo}@us.es*
[b]*Institute of Software Engineering and Programming Languages, University of Ulm, Albert-Einstein-Allee 11, Ulm, 89069, Germany chico.sundermann@uni-ulm.de*
[c]*CRC 1608, KASTEL – Dependability of Software-intensive Systems, Karlsruhe Institute of Technology, Am Fasanengarten 5, Karlsruhe, 76131, Germany kevin.feichtinger@kit.edu*
[d]*Christian Doppler Laboratory VaSiCS, LIT CPS Lab, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria rick.rabiser@jku.at*
[e]*Institute of Software Engineering and Automotive Informatics, TU Braunschweig, Mühlenpfordtstr. 23, Braunschweig, 38106, Germany thomas.thuem@tu-braunschweig.de*

Abstract

Feature modelling is a cornerstone of software product line engineering, providing a means to represent software variability through features and their relationships. Since its inception in 1990, feature modelling has evolved through various extensions, and after three decades of development, there is a growing consensus on the need for a standardised feature modelling language. Despite multiple endeavours to standardise variability modelling and the creation of various textual languages, researchers and practitioners continue to use their own approaches, impeding effective model sharing. In 2018, a collaborative initiative was launched by a group of researchers to develop a novel textual language for representing feature models. This paper introduces the outcome of this effort: the Universal Variability Language (UVL), which is designed to be human-readable and serves as a pivot language for diverse software engineering tools. The development of UVL drew upon community feedback and leveraged established literature in the field of variability modelling. The language is structured into three levels –Boolean, Arithmetic, and Type– and allows for language extensions to introduce additional constructs enhancing its expressiveness. UVL is integrated into various existing software tools, such as FeatureIDE and flamapy, and is maintained by a consortium of institutions. All tools that support the language are released in an open-source format, complemented by dedicated parser implementations for Python and Java. Beyond academia, UVL has found adoption within a range of institutions and companies. It is envisaged that UVL will become the language of choice in the future for a multitude of purposes, including knowledge sharing, educational instruction, and tool integration and interoperability. We envision UVL as a pivotal solution, addressing the limitations of prior attempts and fostering collaboration and innovation in the domain of software product line engineering.

Keywords: feature model, software product lines, variability

1. Introduction

Feature modelling [48], a crucial component of software product line engineering, is one of the most used approaches for representing software variability through the abstraction of features and their relationships [7]. A feature is defined as an increment in product functionality [8]. A software product line is modelled using a *Feature Model* (FM) where features are arranged in a tree-like structure with additional cross-tree constraints. FMs with thousands of features are reported in the literature [15, 53, 69, 85]. FMs are represented using feature diagrams but can also be represented using different textual notations. Textual notations for FMs range from XML–based to tool-specific ones [11]. Over the past thirty years, the evolution of feature modelling has given rise to diverse extensions and representations [26]. However, the absence of a standardised language has impeded effective model sharing among researchers and practitioners, hindering progress in the field.

The year 2018 marked a turning point as a collaborative initiative emerged intending to address the standardisation challenge. This initiative brought together a group of researchers from different universities and research centers under the umbrella of the MODEVAR[1] workshop series. This effort was dedicated to crafting a new textual and simple language for FMs [82]. The outcome of this collective effort is the *Universal Variability Language* (UVL), a solution designed to be both human-readable and a pivot language for a variety of software engineering tools.

UVL's development was not only informed by community feedback but also based on established literature in the field of variability modelling. The language is meticulously structured into three levels –*Boolean*, *Arithmetic*, and *Type*– allowing for a representation of different features and varying types of relationships. Furthermore, UVL embraces extensibility, permitting the introduction of additional constructs to enhance its expressiveness and accommodate diverse modelling needs.

UVL is integrated into existing variability modelling tools, such as FeatureIDE [50] and flamapy [38]. All the tools that support the language are available in an open-source format, complemented by dedicated parser implementations for Python and Java using ANTLR [70][2], which allows designing parsers for other languages. This openness paired with a structured process to involve the community encourages transparency, collaboration, and wider adoption.

We envision an impact of UVL beyond academia, with institutions and companies recognising its potential. As an example, an importer for UVL models

[1] `https://modevar.github.io/`
[2] `https://www.antlr.org/`

was already integrated in a commercial variant management tool [76]. UVL could be used as the language of choice for a myriad of applications, including knowledge sharing, educational instruction, and seamless tool integration. The broad vision for UVL is to overcome the limitations of previous standardisation attempts, such as the Common Variability Language (CVL) [39] or ISO-26558 [46]. CVL [39] eventually and unfortunately failed to become a standard due to legal reasons [72] and ISO-26558 [46] did not reach the community and industry. However, as UVL is community driven, we envision UVL to foster collaboration and innovation within the realm of software product line engineering.

In this paper, we delve into the development, features, and applications of UVL, offering a comprehensive exploration of its significance in the evolving landscape of variability modelling. The contributions of the paper are as follows:

- A tutorial presentation of UVL with a stable version of the language (Section 4) validated by different rounds of participation by the community.
- An extensible language design that provides expressive language features while preserving simplicity with a core language divided into three major levels and an option to decompose large feature models.
- A formal textual syntax and semantics of UVL (Section 5).
- An open source implementation[3] of the language with parsers for Python and Java using ANTLR that allows supporting new general languages such as JavaScript or C# in the future (Section 6).
- A report of our experiences regarding the feasibility of the language based on an interactive and participated process with the community (Section 3) as well as the integration of UVL with different tools (Section 6).

Regarding novelty since previous publications [34, 67, 76, 82, 87], different changes have been introduced and no stable version of UVL was presented so far. The formal syntax of this stable version of UVL as well as the parser implementation supporting Java and Python are new. Furthermore, this work includes the first formal specification and discussion on the semantics of UVL.

The remainder of the paper is structured as follows. Section 2 introduced the necessary background on FMs. We outline the development process and the design goals of UVL in Section 3. After that, we introduce the UVL in a tutorial-like manner and discuss its syntax and semantics in Sections 4 and 5, respectively. We provide an overview of the current UVL implementation and existing tools integrating UVL in Section 6. We then discuss challenges and next steps regarding further adoption of UVL in general and in industry in particular in Section 7. Section 8 concludes the paper.

2. Feature Models

The term *Feature Model* (FM) was coined by Kang et al. in the well–known FODA report in 1990 [48]. Since then, feature modelling has been one of the

[3] `https://github.com/Universal-Variability-Language`

main topics of research in software product lines [37, 35]. There are different FM dialects [81], each with different types of features or relationships, but also with different textual and graphical notations. In the following, we review the most used notations for those languages to pave the way for the presentation of UVL. In general, there is no FM language that can be used in all scenarios and adaptations are often done for concrete domains [6].

A FM is a representation of all possible configurations of a software product line [35]. Given n features, with no restrictions in the combinations of them, 2^n is the number of all potential configurations. With a small n in terms of hundreds, the number of configurations is already very big. An FM restricts this number using feature relationships that represent the constraints of the application domain. FMs are also used in other domains than software product lines such as video encoding [6], biological information [17] or exam options [55] just to mention a few examples. One of the most used examples in the community is the Linux kernel FM, which has thousands of modules and configuration options [88]. Furthermore, large FMs from other domains, such as automotive, with thousands or even tens of thousands of features were reported in the literature [53, 51, 85, 15]. Still, there might be even larger FMs used in practice as FMs from industry are typically not made available.

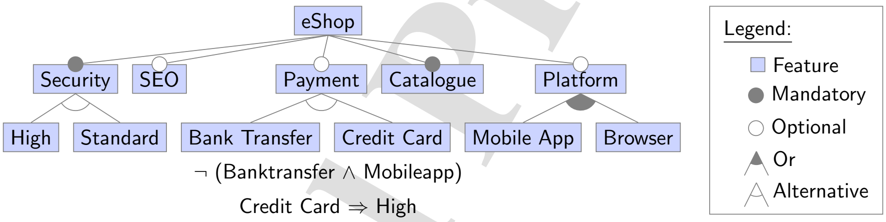

Figure 1: Running example of the online shop case study [75]

Figure 1 shows our running example of the FM for a fictitious `eShop` product line. A FM is composed of a hierarchically arranged set of features (a.k.a. *feature diagram* or *feature tree*) and a set of cross–tree constraints. FMs that do not have cross–tree constraints can exist, but they are not very common. Relationships among features can be of two different types [13]: *i*) relationships between a parent feature and its child features; *ii*) cross–tree constraints that are typically inclusion or exclusion statements or more complex constraints in the form of arbitrary (propositional) formulae.

There are several FM dialects [81]. In this section, we will revisit the most used relationships in the literature and also mention some of the extensions proposed. In basic FMs, the following relationships among features are defined [13]:

- **Mandatory**. A child feature has a *mandatory* relationship with its parent if the child is part of all configurations in which its parent feature is included, e.g., any configuration of an `eShop` has to have a `Catalogue`.
- **Optional**. A child feature has an *optional* relationship with its parent if

the child can optionally be part of all configurations in which its parent feature is included, e.g., a configuration of an `eShop` can optionally have `SEO` support.

- **Alternative**. Child features have an *alternative* relationship with their parent if exactly one of them can be part of a configuration if the parent feature is included. In the example, the *Payment* of the `eShop` must be either `Banktransfer` or `Credit card` (but not both in the same configuration).

- **Or**. Child features have an *or* relationship with their parent if one or more of them can be part of the configuration if the parent feature is included. In Figure 1, whenever `Platform` is selected, `Mobileapp`, `Browser` or any combination thereof including at least one of these two features can be selected.

Note that always a child feature can only be part of a configuration if its parent feature is part of the configuration. Additionally, the root feature is included in all the configurations of the product line. A FM can also contain cross–tree constraints between features – basic ones are the following:

- **Requires**. If a feature A *requires* a feature B, the inclusion of A in a configuration implies the inclusion of B in such a configuration. In the example, an `eShop` including `Credit card` must include `High` security support.

- **Excludes**. If a feature A *excludes* a feature B, both cannot be included in the same configuration, i.e., there is a feature exclusion. The `Banktransfer` feature cannot be combined with a `Mobileapp`, i.e., these two features are incompatible.

More complex cross-tree relationships are often used allowing constraints in the form of generic propositional formulas, e.g., "`A` and not `B` implies `C`" [8]. In some cases, there is a distinction between *concrete* and *abstract* features [90]. Concrete features have a mapping with domain implementation artefacts in the solution space [7], while abstract features are used for organisation purposes and do not have any direct mapping to any artefact in the solution space. Often, only leaves of the tree are concrete features and all the other intermediate features are abstract [9].

2.1. Feature Model Extensions

There are different ways to extend FMs with different constructs. The most well-known families of extensions are *cardinality–based* and *attribute–based* FMs. These extensions include a discussion that has been going on in the community over the years: what are the semantics of feature cardinalities, cloning, or attributes? [18, 20, 27, 74, 80] In this section, we do not repeat such discussions in detail. In following sections when UVL is presented, more details on how those discussions are taken into account will be reported.

Cardinality–based FMs introduce two relationships that resemble those of the *Unified Modelling Language* (UML) with multiplicities in class diagrams – see [19, 74]. The relationships introduced in cardinality–based feature modelling are the following [13]:

- **Feature cardinality.** A feature cardinality is a sequence of intervals $[n..m]$ with n as lower bound and m as upper bound ($n \leq m$). Feature cardinalities are also known as feature clones. The intervals describe the number of instances of the feature that can be part of a configuration. This relationship may be used as a generalisation of the original mandatory ($[1, 1]$) and optional ($[0, 1]$) relationships defined in classical FMs described previously. Cloning a feature means having different instances of the same feature several times in a configuration.
- **Group cardinality.** A group cardinality is an interval $\langle n..m \rangle$, with n being the lower and m the upper bound ($n \leq m$) limiting the number of child features that can be included in a configuration when the parent feature is selected (remember that if the parent is not included in a configuration, none of its children are included). An alternative relationship is equivalent to a $\langle 1..1 \rangle$ group cardinality. An or–relationship is equivalent to $\langle 1..N \rangle$, being N the number of features in the relationship.

Attribute–based FMs. In certain situations, FMs include additional information about the features. For example, the cost or memory consumption of a particular feature in an `eShop` configuration. Such information can be included using *feature attributes*, which are designed for this specific purpose. When FMs are expanded by including additional information in the form of attributes, they are referred to as *extended, advanced, or attribute-based FMs* [6]. Most proposals of attribute–based FMs agree that an attribute should consist at least of a *name*, a *type*, a *domain* and a *value*.

3. Development Process and Design Goals of UVL

We first describe how UVL was developed in a participatory effort in the SPL community and then summarise its design goals.

3.1. Participatory Development Process

UVL is the result of a community effort that started in 2018 as depicted in Figure 2. The idea began with an informal meeting at SPLC 2018 with around twenty key researchers from the SPL community. After a brainstorming session, we agreed on several action points. Among those, it was decided to run a workshop (MODEVAR[4]) to be "an interactive event where all participants shall share knowledge about how to build up a simple feature model language that all the community can agree on".

[4]https://modevar.github.io/

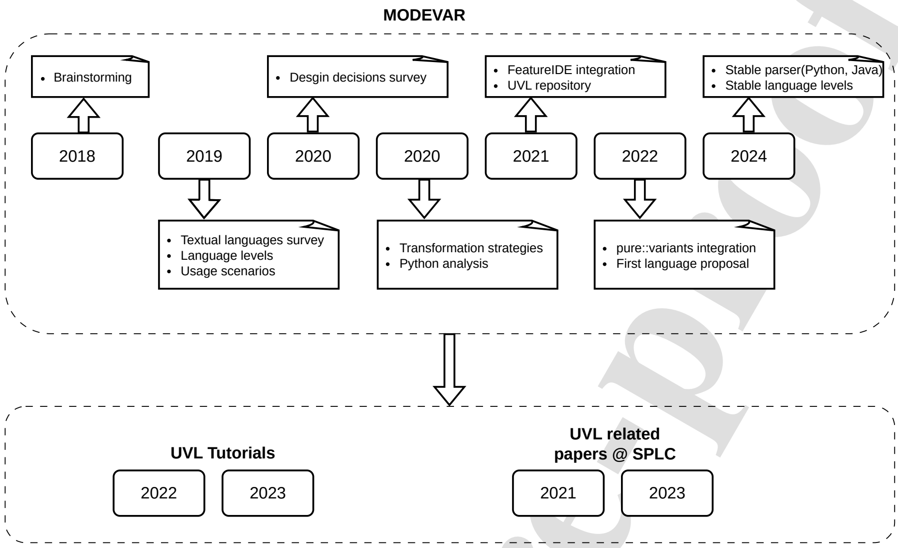

Figure 2: UVL development process since 2018

In the 2019 edition [12], the main outputs were a revisited literature review on textual variability modelling languages [11], the proposal of having different language levels [91] and a set of fourteen usage scenarios of the language [14] described by examples. Those scenarios were the result of a systematic process, where members of the community gave original descriptions, which received feedback via a survey and expert feedback. The survey, the language levels, and the usage scenarios were used for the next steps in the process.

During the 2020 VaMoS event, a survey (results later published at SPLC 2021 [82]) aimed at informing the language's design decisions was performed. This survey comprised a questionnaire administered to 20 workshop attendees. In the initial part of the questionnaire, participants shared their preferences concerning the gathered structural attributes of the language. Subsequently, in the latter part, attendees deliberated on which language features should be incorporated based on their considerations. Throughout the questionnaire, participants collaborated in pairs to deliberate on their viewpoints and offer more meticulously considered responses.

The workshop was run again in 2020 at SPLC [2] (online due to the pandemic). Transforming different variability models is a challenge, in [32] usage scenarios, required capabilities and challenges for an approach for semi-automatically transforming variability models were presented. One of the conclusions was that a pivotal common language can help to transform variability artifacts and underlines the necessity of UVL in this sense as a pivotal language for transformations. In addition, a new tool based on Python to analyse FMs was presented [36] with the potential of including UVL as variability language (see Section 6.2.3).

In 2021 [89], two concrete integrations of UVL in different tools were presented. First, an integration of one of the previous versions of UVL in FeatureIDE showed the feasibility of the language [84]. Second, a prototype of a repository to share UVL models was presented [77]. This way, some of the objectives of the language started to materialise: tools integration and knowledge sharing (see Section 6).

In 2022 [45], another integration, in this case with a commercial variant management tool was presented [76]. Concretely, UVL was integrated with pure::variants [71], one of the most well-known commercial tools in the software product line engineering area. In addition, a first tutorial on a previous version of UVL was given.

During these years, some tools integrated different versions of UVL producing their own parsers [38, 57, 41]. In 2023, there was an implementation effort to produce a common stable parser of UVL with support for Python and Java. This parser was briefly introduced during the MODEVAR 2024 edition at VaMoS 2024 and it is one of the contributions of this paper (cf. sections 4 and 6). Additionally, further developments around UVL and its expressiveness were presented [5, 43, 78].

In 2024 a second MODEVAR edition took place [30]. This time the focus of the community turned towards the adoption of UVL in industry. For that, potential challenges [73] and necessary extensions [29] for UVL were discussed with a representative of pure::variants [71]. Additionally, a generator for UVL models in arbitrary size and complexity was introduced, which facilities scalability analysis [86].

Although MODEVAR has been the meeting point of researchers and practitioners with interest in the development of a simple, common textual feature modelling language, the outputs and discussions of the workshop served to produce other artefacts outside of the workshop [52, 83]. Concretely, there were two tutorials at SPLC 2022 and 2023 presenting the advances of UVL as well as analysis and transformation capabilities. Also, there were two major papers at SPLC 2021 and 2023 presenting a first version of the language [82] and some transformation and analysis capabilities [87].

In summary, one unique selling point of UVL is the community-driven design of the language [82]. With various surveys and discussions with experts of the community, different authors derived requirements for the design of a widely adopted variability language [11, 12, 14, 82, 91]. In the following, we present derived requirements that influenced the language design and how we address these in UVL.

3.2. Design goals

Designing a language is difficult [92]. With the participatory process described in the previous section, we mitigated the possibility of having a language that was not accepted by the community. With the inputs of the workshops and working sessions, we defined several design goals that are summarized as follows:

Simplicity. In general, UVL should be simple to use. For simplicity, we consider two dimensions: (1) UVL should be easy to use, understand (with simple constructs), learn and comprehend (facilitating the comprehension of the variability in hand) for humans [14] and (2) it should not require too much effort to integrate UVL in variability modelling tools. For human understandability and comprehension, UVL should use concepts familiar to users. As potential users, we consider people working in the computer science field and/or using variability modelling. Hence, we aim to use concepts from programming languages, modelling in computer science (e.g., grammars or meta-models), and existing variability languages. For easier integration, we consider the following requirements for UVL. First, the core language should be simple so that developers do not have to integrate various complex constructs. Second, the language should reuse existing concepts from other variability languages, e.g., common keywords like *alternative*. Third, the core language should be simple to analyse with conventional analysis tools used in the domain, such as SAT [22, 65, 93], BDD [40] or #SAT solvers [53, 56, 85].

Information Hiding. In practice, it often makes sense to only work on small subparts of a variability model. First, large variability models are typically hard to oversee [3]. Second, different stakeholders commonly do not work on the entire variability model but specific parts [3]. Hence, UVL should have a mechanism to support focusing on a subset of interest.

Expressiveness. To be widely applicable, one of UVL's goals is to cover many practical use cases. First, users of UVL should be able to specify constraints as needed to describe the set of valid configurations, which may include propositional logic, constraints over numeric values, or even reasoning about content of strings [11]. Second, UVL should be able to describe constructs used in available feature modelling tools [4, 38, 61, 64].

Extensibility. A higher expressiveness conflicts with the goal of simplicity [91], as more and potentially more complex language constructs need to be supported. As a compromise in UVL, we aim to have an extensible language design with a simple core language that can be easily adopted and extensions that introduce more expressiveness. Here, we use the concept of language levels [91] that encapsulate different language constructs and extend the UVL core language.

Exchange. Models of a common variability language should be exchangeable between different tools [14]. For simplifying exchange, we consider two aspects. First, available tool support (e.g., for parsing) should be reusable for different users of UVL. Second, there should be a mechanism to exchange UVL models between tools that support different levels of expressiveness.

4. The Universal Variability Language (UVL)

In this section, we illustrate how to specify variability models with UVL[5] using our running example. The design of UVL consists of a simple base language with several language extensions, which we call language levels (cf. Section 5). Here, we start with a simple version and extend it iteratively to showcase more expressive UVL language levels. For a formal description on the language, we refer to Section 5.

4.1. Language Levels

In UVL, we use *language levels* to tackle expressiveness and extensibility while preserving a simple core language. The idea is that users of UVL can limit their models to specific language constructs. If a tool only supports very simple constructs, higher language levels can be forbidden. If more expressiveness is needed, additional language levels can be enabled.

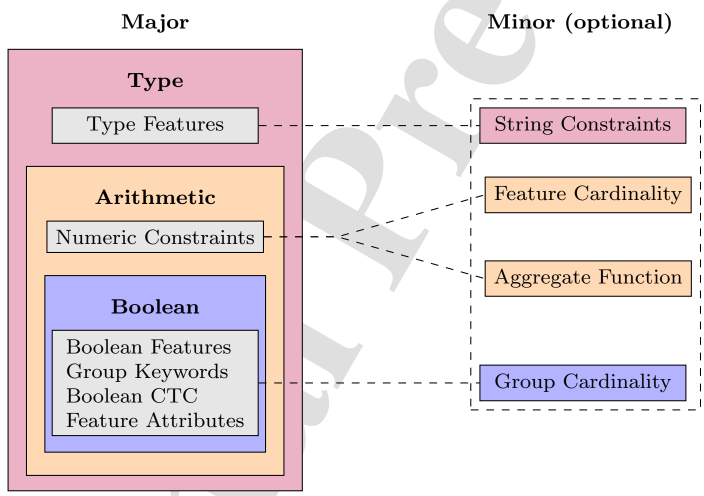

Figure 3: Language Level Hierarchy in UVL

Figure 3 shows the language levels currently available in UVL. Each language level encapsulates certain language constructs. We distinguish between *major* and *minor* language levels. The major levels have a hierarchical order. The *Boolean*-level is the *core* language of UVL. The *Arithmetic*-level fully includes the major Boolean level and extends it with numeric constraints over feature attributes. The *Type*-level extends both with typed features, such as string or numeric features. The goal of these levels is to separate the language according to reasoning engines that could be used to reason about them. For instance, the *Boolean*-level can be simply encoded as a SAT problem. *Minor* language levels

[5]We discussed different name alternatives for the proposed language and we decide to use UVL because the intention is to make it an *Universal* language used by many stakeholders in the variability modelling community.

are optional extensions of the major levels. The idea is to separate constructs that can be analysed with the same reasoning engine but may require further handling or are not always supported by available tools. They are not automatically included in higher language levels. *Group Cardinality* extends the boolean level with cardinality group relationships which enable selecting [n..m] features from the group. *Feature Cardinality* and *Aggregate Functions* are optional extensions of the arithmetic level. They enable (1) selecting a feature multiple times and (2) aggregates, such as sums, over numerical attributes, respectively. *String Constraints* add constraints to compare strings and lengths of strings. In the following, we showcase the different language levels by extending our base example shown in Listing 1.

4.2. Boolean Level

Listing 1 shows our running example from Figure 1 in `UVL` syntax. The `UVL` model consists of two main parts: the feature tree and the cross-tree constraints. The tree hierarchy is represented using indentation. Keywords are used to specify the parent-child relationship. As in Figure 1, `eShop` has two mandatory and three optional child features. Furthermore, exactly one `Security` option and exactly one `Payment` option can be selected as denoted by the *alternative*-keyword. For the feature `Platform`, the *or*-group denotes that at least one platform can be selected. The cross-tree constraints are used to impose further limitations on the features. For instance, `Bank Transfer` and `Mobile App` cannot be included in the same configuration, i.e., they are incompatible features. Also, a `Credit Card` requires a `High` security level. For the core language, the constraints are limited to propositional logic.

In addition to feature dependencies, the `UVL` model contains some attributes that provide information on the respective features. In our model, we have a number attribute (price), a Boolean attribute (SEPA), and a string attribute (URL). In the core language of `UVL`, attributes can only be used for storing information about features that do not influence the validity of configurations. Constraints over attributes are excluded in the core language, since the reasoning is considerably more complex and not straightforward to encode for many automated reasoning engines, such as SAT solvers. Still, attributes are relevant to (1) store tool-specific information, (2) attach general information to features, and (3) can be used to compute metrics for configurations based on user selections, such as a price.

For further information, comments can be added either single line with `//` or multiple lines with `/* <comment> */`. All comments are discarded during the parsing process.

Listing 2 shows an adaptation of the previous `eShop` but using now the cardinality capacity for a feature group. Another change is the *include* at the very top of the listing. The include mechanism allows users to specify explicitly which language constructs are supported. This can be used for (1) providing information on the contents of the language level and (2) ensure that users do not introduce constructs that are not supported by the tool using `UVL`. In the latter case, the `UVL` parser should provide information on the mismatch of

Listing 1: UVL Running Example: Core

```
features
    eShop
        mandatory // select all
            Security
                alternative // select exactly one
                    High {Price 100}
                    Standard {Price 50}
            Catalogue
        optional
            SEO
            Payment
                alternative
                    "Bank Transfer" {Price 10, SEPA true}
                    "Credit Card" {Price 20}
            Platform
                or // select at least one
                    "Mobile App"
                    Browser {URL 'www.uvleshop.org'}

constraints
    !("Bank Transfer" & "Mobile App")
    "Credit Card" => High
```

declared and used levels. By default, i.e., when no language levels are specified in includes, all language levels are included. Each construct in the initial `eShop` Listing 1 is part of the core language. Group cardinality is a *minor* level of the Boolean (i.e., core) language level. Including the minor level group-cardinality automatically includes its major level Boolean. In Section 4.3 and Section 4.4, we illustrate the other two major language levels in UVL, namely *Arithmetic* and *Type*, using our running example.

Listing 2: UVL Running Example: Group Cardinality

```
include
    Boolean.group-cardinality

features
        ...
            Platform
                [2..3]
                    "Desktop App"
                    "Mobile App"
                    Browser {URL 'www.uvleshop.org'}
        ...
```

Group Cardinality. In addition to the specification of included language levels, there are changes in the feature tree. Now, in Listing 2 the customer has three `Platform` options to choose from. In addition, the group-type changed to a *group cardinality*. The group denotes that the customer needs to select between two and three ([2..3]) platform features instead of at least one.

4.3. Arithmetic Level

In this section, we extend our FM with constructs from the *Arithmetic*-level and its minor levels. Listing 3 further enriches our `eShop` with an *arithmetic constraint* over the price attribute. The constraint denotes that the overall sum in price of all selected features should be smaller than 200. With the *Arithmetic*-level, the following operators are supported: +, -, *, /, ==, <, >, <=, and >=. The minor level *aggregate-function* also introduces sum() and avg().

Listing 3: UVL Running Example: Arithmetic

```
include
    Boolean.group-cardinality
    Arithmetic.aggregate-function

features
        ...

constraints
    !("Bank Transfer" & "Mobile App")
    "Credit Card" => High
    sum(Price) < 200
```

Feature Cardinality. In Listing 4, we introduce *feature cardinality*, which is a minor level of the *Arithmetic*-level. In our example, the user can decide to have between one and five `Catalogue` features as denoted by *cardinality* [1..5]. A customer may select varying catalogues for different markets, e.g., Europe and North America. Note that each selected `Catalogue` would increase the overall price by 30.

Listing 4: UVL Running Example: Feature Cardinality

```
include
    Boolean.group-cardinality
    Arithmetic.aggregate-function
    Arithmetic.feature-cardinality

features
    eShop
        mandatory
            Security
                alternative
                    High {Price 100}
                    Standard {Price 50}
            Catalogue cardinality [1..5] {Price 30}
    ...
```

4.4. Type Level

Listing 5 shows the last version of our `eShop` with all language levels included. Here, we newly added the *Type*-level, which introduces features with the following types: integer, float, and string. Note that any feature can still be deselected even if it is not Boolean. For, instance the customer can now

Listing 5: UVL Running Example: Typed Features

```
include
    Boolean.group-cardinality
    Arithmetic.*
    Type.string-constraints

features
    eShop
        mandatory
            Security
                alternative
                    High {Price 100}
                    Standard {Price 50}
            Catalogue cardinality [1..5] {Price 30}
            Integer "Items in Basket"
        optional
            SEO {Price 40}
            Payment
                alternative
                    "Bank Transfer" {Price 10, SEPA true}
                    "Credit Card" {Price 20}
            Platform
                [2..3]
                    "Desktop App" {Price 70}
                    Boolean "Mobile App" {Price 80}
                    String Browser {Price 20}

constraints
    !("Bank Transfer" & "Mobile App")
    "Credit Card" => High
    sum(Price) < 200
    0 < "Items in Basket"
    len(Browser) < 30
```

configure an integer feature `Items in Basket`, which can be used to limit the maximum number of items a customer can put in his basket at the same time. A cross-tree constraint ensures that the maximum number of items is higher than zero. Further, `Browser` is now a string feature where the URL can be directly configured. Another cross-tree constraint denotes that the URL may not be longer than 30 characters. The used *len*-function is part of the *string-constraints* minor level which also introduces equality checks between strings. Note that we also replaced the two lines for specifying both minor levels of the *Arithmetic* level with a wildcard `Arithmetic.*`.

4.5. Import Mechanism

With thousands of features and constraints in practice [58, 53, 85], FMs are often hard to overview. Further, stakeholders often only need to consider a subset of the FM. To simplify managing large FMs and focusing on parts of interest, UVL provides a mechanism for decomposing models into subparts that can then be imported in an overall model if needed.

Listing 6 showcases the *import* mechanism of UVL where we have the `Platform` subtree (Listing 7) and the `Security` subtree (Listing 8 as separate files. Those subtrees are imported using the *imports*-keyword. Imports are specified using a relative file path to the imported UVL model. For instance, `platform` refers to a file in the same directory named `platform.uvl`. Non-trivial paths can be specified with a Python-like dot notation (e.g., submodels.platform). Imports can also be given an alias with the *as* keyword. The submodel can then be attached to an arbitrary location in the feature tree by referencing its root feature (e.g., `pl.Platform`). In the cross-tree constraints all features of submodels can be referenced using the submodels' namespace. The shown model (Listing 6) is equivalent to Listing 5. Semantically, the feature reference in the composed model is expanded to include the entire subtree. For instance, `pl.Platform` references the entire FM in Listing 7. Also, all cross-tree constraints in the imported submodels are applied for the composed model. Cross-tree constraints in the composed can reference features from imported submodels using the filename or alias and the feature name. For example, in line 21 `pl."Mobile App"` is referenced. The import mechanism may have the following two advantages for our running example. First, Listing 6 is shorter and easier to overview than the entire model shown in Listing 5. Second, a developer only responsible for platform or security development can separately work on the submodels Listing 7 and Listing 8, respectively.

Listing 7: UVL Running Example: Platform Submodel

```
features
    Platform
        [2..3]
            "Desktop App" {Price 70}
            Boolean "Mobile App" {Price 80}
            String Browser {Price 20}

constraints
    len(Browser) < 30
```

Listing 8: UVL Running Example: Security Submodel

```
features
    Security
        alternative
            High {Price 100}
            Standard {Price 50}
```

Listing 6: UVL Running Example: Import Mechanism

```
imports
    platform as pl
    security

features
    eShop
        mandatory
            security.Security
            Catalogue cardinality [1..5] {Price 30}
            Integer "Items in Basket"
        optional
            SEO {Price 40}
            Payment
                alternative
                    "Bank Transfer" {Price 10, SEPA true}
                    "Credit Card" {Price 20}
            pl.Platform


constraints
    !("Bank Transfer" & pl."Mobile App")
    "Credit Card" => security.High
    sum(Price) < 200
    "Items in Basket" > 0
```

Summary. UVL provides a simple core language and an import mechanism that enables decomposing models into manageable small submodels to tackle its design goal *simplicity*. Additional language levels provide more *expressiveness* with constructs to specify cardinalities, different constraints over numeric values, typed features, and constraints over strings. The design of the language levels is *extensible* to allow different users to tailor their UVL models to their use case and tool limitations. We used this section to introduce UVL with an example, in Section 5 we define UVL models more formally and discuss the semantics of different constraints.

5. Syntax & Semantics: Language Specification

In this section, we discuss the syntax and semantics of UVL more formally. The goal is to clarify possible ambiguities and provide clear guidelines on how to interpret UVL and work with the language. Note that we use some concepts here requiring computer science background to understand.

5.1. UVL Syntax

Figure 4 shows a simplified view on the abstract syntax of a UVL model in form of a meta-model (more details on concrete parts of the meta-model will be elaborated later). A UVL model consists of four major parts: *imports*, *language levels*, *feature tree*, and *cross-tree constraints*. In the following, we explain the four major parts in more detail and the language constructs that can be used within.

Imports. As discussed in Section 3, decomposing a feature model into smaller sub-parts is beneficial. Still, knowledge about cross-dependencies between those sub-parts needs to be maintained as they may impact the configuration space. With UVL, we support composition of various smaller sub-models with an import mechanism. Hereby, another UVL model can be imported via `import submodel`. Then, the submodel can be referenced at an arbitrary location in the feature tree with `submodel.Root`. Note that `Root` is the name of the root feature here. While the composed model only contains one line for adding the root feature, this is semantically equivalent of copying the entire sub-model at this location. Cross-tree constraints of the sub-model also apply for the composed model. Constraints between features of different sub-models can be specified using the same syntax as in the feature tree (e.g., `submodel1.A & submodel2.B`). For each import, an alias can be specified with the *as*-keyword (e.g., `import submodel1 as s1`). Features can then be referenced with `s1.A`. Submodels in other, possibly nested, directories can be referenced with `<dir1>.<dir2>.<uvlfile>`.

Language Levels. Language levels can be explicitly specified with the *include* keyword. The included language levels are listed in separate lines using the syntax `<major>.(<minor>|*)?`. So, one line can either specify a major level (`<major>`), a minor level (`<major>.<minor>`, or all minor levels (`<major>.*`). If a developer violates the language levels by adding an unsupported construct, the

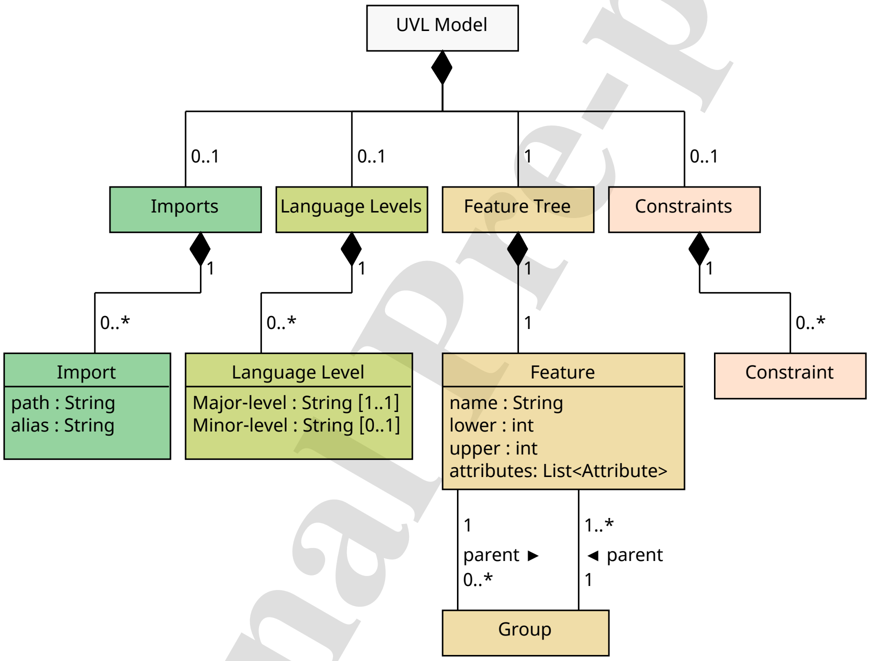

Figure 4: UVL Meta Model

parser would provide a warning or error to him. Note that using a minor level always includes the respective major level. By default, all language levels are included. Hence, not specifying any level includes enables the full expressiveness of UVL.

Figure 3 shows the language levels currently supported in UVL and the language constructs they include. The three major levels *Boolean*, *Arithmetic*, *Type* encapsulate language constructs that can be reasoned about with a specific reasoning engine. For instance, UVL models of the Boolean level should be straightforward to encode as a SAT instance (e.g., CNF)[8, 54]. In contrast, the Arithmetic level can be directly represented as SMT [24] or CP [47] problem, but requires further processing to be encoded as SAT instance.

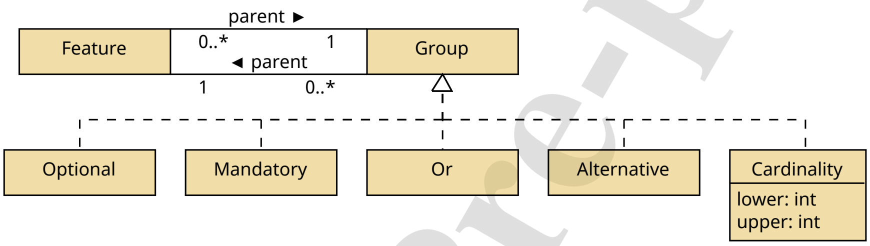

Figure 5: UVL Feature-Group Types

Feature Tree. The UVL feature tree consists of two main elements: *features* and *groups*. The tree requires exactly one root feature. Each feature may have an arbitrary number of groups which in turn may have an arbitrary number of features each. The relationship between features and groups are denoted with indentation. For a feature, its corresponding groups are indented by one line and vice versa for groups. A feature always requires a unique *name* as identifier. Here, the identifier needs to be enclosed by quotation marks if the used symbols may introduce an ambiguity in the UVL model.[6] Each feature can have a *feature cardinality* [n..m], which denotes that the feature can be selected between n and m times. Also, a list of *attributes* {att1, att2, ...} can be attached. There are five feature group types supported in UVL as seen in Figure 5. *Optional*, *mandatory*, *or*, and *alternative* are part of the core (Boolean) level, while *group cardinality* is a Boolean minor level.

Feature Attributes. For each feature, an arbitrary number of attributes can be attached. Generally, attributes are key-value pairs with the key being an identifier and the value being of one of the types shown in Figure 6. One exception is that it is allowed to only specify a key, which is then considered as Boolean attribute with true as value. The attributes of a feature are declared in curly brackets as follows: {<key1> <val1>, <key2> <val2>}. Nested *attribute lists* can be specified with {<key> {<key1> <val1> ...}}. Types of attributes

[6] Identifiers not matching [a-zA-Z0-9_]*[a-zA-Z_][a-zA-Z0-9_]* must be protected.

are not explicitly stated but rather inferred from the value. String constants (i.e., values of string attributes) are specified with single quotation marks to prevent ambiguities with feature names.

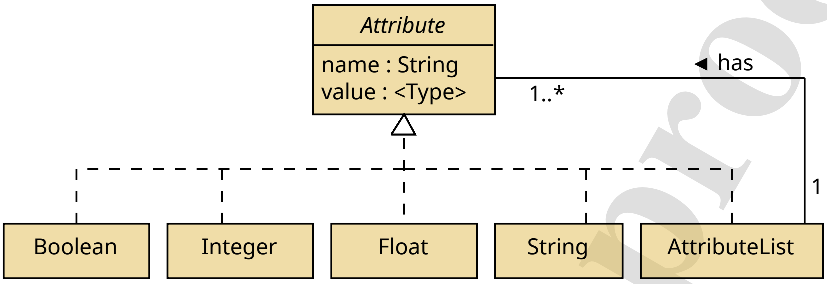

Figure 6: UVL Feature Attributes

Constraints. The constraints part of a UVL model consists of a list of constraints that can evaluate to either true or false. A valid configuration needs to satisfy every attached constraint. In UVL, constraints are mostly based on common propositional logic operators, namely ! (not), & (and), | (or), => (implies), <=> (equals), and brackets. These operators combine Boolean variables, Boolean constants (i.e., true or false), or predicates(cf. Figure 8), which each need to evaluate to a Boolean. The Boolean variables can refer to either a feature name or an Boolean-attribute key.

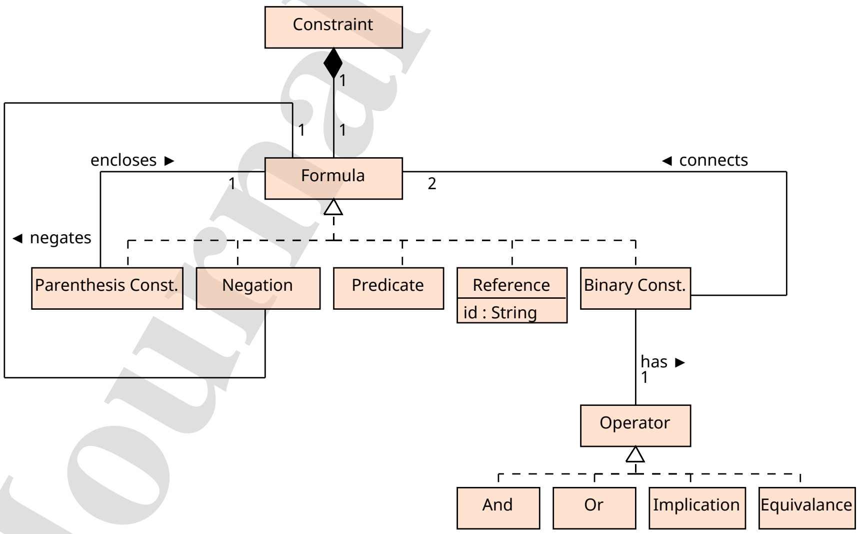

Figure 7: UVL Cross-Tree Constraint

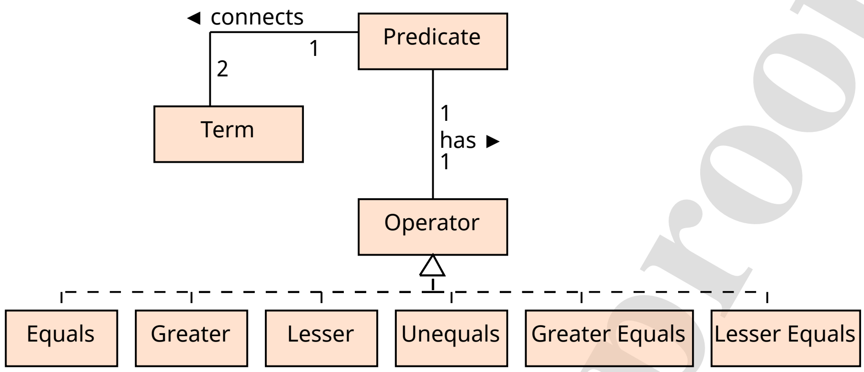

Figure 8: UVL Predicates

Predicates. Predicates are used in UVL to specify dependencies based on string and numerical values. As shown in Figure 8, each predicate has exactly one operator of equals (== in UVL), greater (>), lesser (<), unequal (!=), greater equal (>=), and lesser equal (<=). Note that each of these operators evaluates to a Boolean value. Those operators connect two terms, which are illustrated in Figure 9.

Terms can only be used within predicates in UVL. A term can be (1) a reference to a variable, (2) a constant, (3) a function, or (4) a binary expression. The referenced variables can be either features or attributes. Constants can be either strings or numeric values. String constants are always enclosed with single quotation marks to prevent ambiguities with references. Otherwise, it would be impossible to distinguish between a string constant matching a feature name and said feature. For functions, *sum*, *average* are currently supported for numeric values and *length* for strings. The binary expression can connect two numeric values with simple arithmetic operators, namely add (+ in UVL), subtract (-), multiply (*), and division (/).

5.2. Constraint Semantics

In this section, we discuss the semantics of constraints in UVL considering on how they affect the set of valid configurations. Our goal here is to clarify potential ambiguities in the semantics of specific constraints. Table 1 shows a formal definition of the restrictions different constraints impose on the *configuration space* VC (i.e., the set of valid, complete configurations modeled by a UVL model). For $C = (I, E)$, I is the set of included features and E of excluded features. Further, f is a feature, $p(f)$ the parent feature of f, $s(f, C)$ the selection status of f in C, card(f, C) the cardinality of f in C. The selection status is a function s : (feature, configuration) $\rightarrow \{0, 1\}$ that maps a feature selected (1) or deselected (0). The cardinality of a feature card(f, C) describes the selection of a feature as integer number. Note that features without denoted cardinality

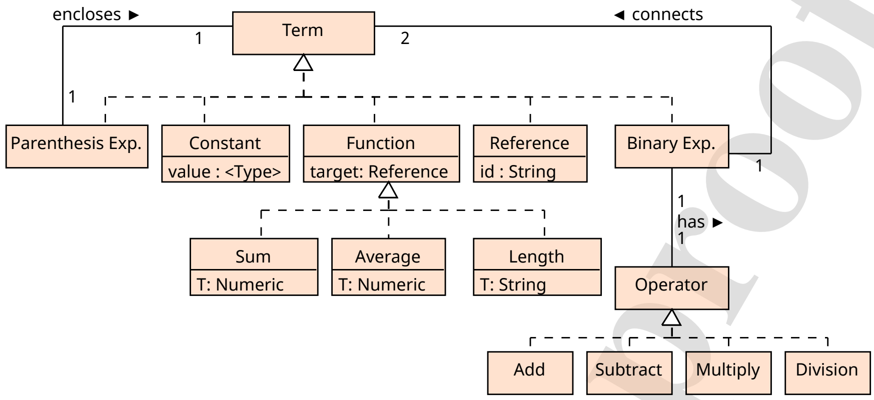

Figure 9: UVL Terms

can only have the values 0 and 1. G is a set of features and ϕ is an arbitrary logical formula. The semantics of the different constraints are equivalent to the descriptions in Section 2.

Feature Cardinality. The semantics of feature cardinality are not straightforward and drive discussions in research [21]. Generally in UVL, we consider subtrees induced by a cardinality as clones that can be configured separately. However, the current syntax does not support referencing specific clones, which requires a default behaviour in handling feature cardinalities in other constraints. In the following, we discuss how we interpret interactions between feature cardinality and cross-tree constraints. We use Listing 9 to illustrate the interactions of feature cardinalities within a simple UVL model. The feature `A` can be selected between two and three times. As a consequence, the entire subtree including `B` and `C` can be configured up-to three times. However, it is not straightforward to interpret both cross-tree constraints. If we select `A` two times and also select `D`, do we need to select `C` for every subtree clone as consequence of the implication $D \Rightarrow C$? Or do we need to select at least one `C`? In the following, we explain on how we interpret feature cardinalities with UVL.

Listing 10 shows a UVL model where we resolved the feature cardinality to illustrate its semantics. The feature cardinality consists of three clones of the original subtree within a group cardinality that ensures that [2..3] of those subtrees must be selected. Listing 10 also depicts three variants to interpret the cross-tree constraints. The first version *contextual clone constraints* is the interpretation used in UVL. Here, we have copies of the cross-tree constraints containing a feature from the feature cardinality subtree for each clone (i.e., A_1–A_3). Each of those constraints is only applied in its respective context (i.e., its subtree is selected). For instance, (`B_2 => C_2`) only needs to be satisfied when the second instance of `A` is selected. This is realised in UVL with the implication `A_2 => (B_2 => C_2)`. If the constraint would always be applied (i.e., only having

Table 1: Constraint Semantics

Constraint	If $C = (I, E) \in VC$ this needs to hold
Feature f	$s(p(f), C) \geq s(f, C)$
Root f	$s(f, C) = 1$
Mandatory f	$s(p(f), C) = s(f, C)$
Group Cardinality [n..m] G	$n \leq \sum_{f \in G} s(f, C) \leq m$
Alternative G	$\sum_{f \in G} s(f, C) = 1$
Or G	$\sum_{f \in G} s(f, C) \geq 1$
Feature Cardinality [n..m] f	$n \leq \text{card}(f, C) \leq m$
Cross-tree constraint ϕ	$\text{SAT}(\phi \wedge \bigwedge_{i \in I} i \wedge \bigwedge_{e \in E} \neg e)$

Listing 9: Cardinality Interactions in UVL

```
features
    R
        optional
            A cardinality [2..3]
                optional
                    B
                    C
            D

constraints
    B => C
    !D | (C & B)
```

right side of implication), the constraints would automatically apply for every clone. In particular, when `D` is selected, it would be required to select every `C_i` and in consequence every `A_i`. Hence, selecting `D` would enforce selecting three instances of `A`. In addition to the contextual clone constraints, we have one copy of constraints containing features that are part of the cardinality subtree and ones that are not. Here, we replace each occurrence of a cardinality feature f with an or over each of the clone features $f_1 \vee \ldots \vee f_m$. For an example, see the first cross-tree constraint in Listing 10. The idea of this constraint is to ensure constraints with other features are met with at least for one clone. We assume that this often matches the expectation for constraints such as `D => C`, where one would expect that selecting `D` requires to have a `C`.

Listing 10: Cardinality Semantics Different Versions

```
features
    A
        [2..3]
            A_1
                optional
                    B_1
                    C_1
            A_2
                optional
                    B_2
                    C_2
            A_3
                optional
                    B_3
                    C_3
    D

// Contextual Clone Constraints
constraints
    !D | ((C_1 | C_2 | C_3) & (B_1 | B_2 | B_3))
    A_1 => (D & (C_1 | B_1))
    A_2 => (D & (C_2 | B_2))
    A_3 => (D & (C_3 | B_3))
    A_1 => (B_1 => C_1)
    A_2 => (B_2 => C_2)
    A_3 => (B_3 => C_3)
```

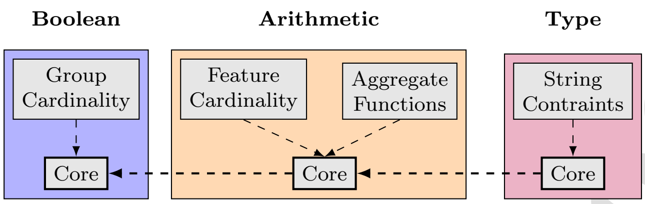

Figure 10: Language Levels in UVL

5.3. Conversion Strategies

With the extensible language design of UVL, another problem arises: the *exchange* of UVL models between tools that employ different language levels. If tool A supports a higher language level than tool B, feature models of tool A cannot be used in tool B. This may even be an issue for a single developer, as different variability modelling tools have different capabilities and advantages [62, 4, 64, 57].

With UVL, we tackle the issue of *exchanging* models with different language levels by using *conversion strategies*. Note that we only consider translation between different levels of UVL here in contrast to conversions to other variability languages as performed by tools such as `TraVarT` [34]. Figure 10 shows the current language level hierarchy of UVL and conversion strategies between them. A conversion takes a UVL model of a certain level and converts it to a UVL model of the next lower level by replacing the constructs with semantically equivalent constructs from the lower level. The possible conversions and their directions are marked in Figure 10 with dashed arrows. For a minor level, its next lower level is its major level (e.g., Boolean for group cardinality). For a major level, its next lower level is the most expressive major language level that is *included* (cf. Figure 3) by the level to translate. In Figure 10 the hierarchy is illustrated from right to left with the rightmost (Type) having the highest hierarchy. Since we have a conversion to the next lower level for every level, we can transitively convert higher language constructs to lower ones. Also, we only need to implement one additional conversion when introducing a new language level. Note that models may exponentially grow in size when converted.

Table 2 shows the conversion strategies applied in UVL. The rows show the source level, the target level when converting the source level, an illustration of the construct in the source level, and the result of the respective conversion. For *Group Cardinality* and *Arithmetic*-level, we provide a specific example in the table for comprehensibility of the conversion strategy. For conversions of both concepts, we specify the valid partial configurations over the features involved according to the respective constraint. Involved are the features in the Group Cardinality and the features occurring in the predicate, respectively. For *Feature Cardinality*, we expand the Feature Cardinality by introducing the respective number of clone (cf. Section 5.2). Note that we directly convert the Group Cardinality according to the presented conversion strategy, since conversions do not rely on other minor levels. For *Aggregate Functions*, we expand the function by applying the operation on all features that have the respective attribute. For the *Type*-level, we transform the features with different types to a boolean feature which has an attribute according to the type of the original feature. Currently, we just drop the string constraints from the *Type*-level, since we found no way to represent those constraints with constructs from other levels.

Table 2: Conversion Strategies

Source	Target	Original	Converted
Boolean.group-cardinality	Boolean	[2..3] f_1 f_2 f_3	$(f_1 \wedge f_2 \wedge \neg f_3) \vee (f_1 \wedge \neg f_2 \wedge f_3)$ $\vee(\neg f_1 \wedge f_2 \wedge f_3)$
Arithmetic	Boolean	f_1.a + f_2.b + f_3.c < 20 f_1.a = 5, f_2.b = 10, f_2.b = 16	$\neg f_1 \vee \neg(f_2 \wedge f_3)$
Arithmetic.feature-cardinality	Arithmetic	a cardinality [n..m]	a [n..m] a_1 ... a_m
Arithmetic.aggregate-function	Arithmetic	sum(a) avg(a)	$f_1.a + f_2.a + ... + f_n.a$ $\frac{(f_1.a+f_2.a+...+f_n.a)}{f_1+f_2+...+f_n}$
Type	Arithmetic	Integer f Float f String f Boolean f	f {Integer 0} f {Float 0} f {String 0} f
Type.string-constraints	Type	f == "Fun" len(f)	Drop Drop

6. UVL Implementation and Integration with other Tools

In this section, we introduce the reference parser implementation of the UVL language. Further, we outline other tools integrating the UVL for various purposes, including graphical editing, textual editing, analysis, configuration, and transformation.

6.1. UVL Parser

The current UVL parser [87] extends the previous implementations by the presented language levels discussed in the previous sections. The parser is based on ANTLR [70] and is available as an open-source implementation in Java and Python.[7] Listing 11 shows the syntax of a UVL model implemented in the parser as a simplified grammar in an EBNF-like notation. The parser implements the necessary conversion strategies between *Boolean*, *Arithmetic*, and *Type*-level of UVL (cf. Table 2). A conversion only works from a higher level to the next lower level to allow importing more expressive UVL models in tools that only build on lower levels. Thus, concepts of the *Type*-level are converted to the *Arithmetic*-level and these concepts are then converted to the core *Boolean*-level. Figure 10 shows the implemented language concepts, organised per level and highlighting the conversion strategies. In previous work, we implemented parsers based on Clojure [82] and Python [36], but both those parsers are limited to the Boolean level of UVL.

6.2. Available Tooling

Several tools, such as FeatureIDE [61, 84], flamapy [38], TRAVART [34], or variability.dev [43] have integrated UVL for different purposes. Most tools either enable graphical [43, 61, 84] or textual [57] editing, analysis [38, 43, 57, 61, 67, 84], transformation [34, 67], or configuration [38, 43, 57, 61, 84] of UVL models. Some tools, such as FeatureIDE [61, 84], flamapy [38] or Nemo [67] do support multiple purposes at once. For instance, the UVLS [57] provides an implementation of the language server protocol to enable easy integration into existing tools and additionally the configuration of UVL models. Other tools use UVL to facilitate variability model interoperability via the transformation of variability artefacts into UVL [34, 67]. Further, some tools, such as pure::variants [71, 76], ddueruem [42], FM Fact Label [44], or V4rdiac [28], integrated UVL or one of the tools supporting UVL to expand the range of supported variability artefacts in their respective tool. Not all tools support all language levels of UVL. In the following, we discuss the respective tools, their focus, and which UVL language level they support. Table 3 summarises the discussed tools.

6.2.1. Graphical editing

FeatureIDE [61, 84] is the de-facto standard for graphical editing of feature models. The Eclipse-based tool allows defining feature models using the core of the UVL *Boolean*-level. Thus, UVL models created in FeatureIDE may consist of optional and mandatory features, and each feature may consist of a set

[7] UVL Parser – `https://github.com/Universal-Variability-Language/uvl-parser`

Listing 11: Simplified UVL Grammar in EBNF Notation

```

featureModel: includes? NEWLINE? imports? NEWLINE? features? NEWLINE? constraints? EOF;

includes: 'include' NEWLINE INDENT includeLine* DEDENT;
includeLine: languageLevel NEWLINE;

imports: 'imports' NEWLINE INDENT importLine* DEDENT;
importLine: ns=reference ('as' alias=reference)? NEWLINE;

features: 'features' NEWLINE INDENT feature DEDENT;

group
    : ORGROUP groupSpec          # OrGroup
    | ALTERNATIVE groupSpec # AlternativeGroup
    | OPTIONAL groupSpec    # OptionalGroup
    | MANDATORY groupSpec   # MandatoryGroup
    | CARDINALITY groupSpec    # CardinalityGroup
    ;

groupSpec: NEWLINE INDENT feature+ DEDENT;

feature: featureType? reference featureCardinality? attributes? NEWLINE (INDENT group+ DEDENT)?;

featureCardinality: 'cardinality' CARDINALITY;

attributes: OPEN_BRACE (attribute (COMMA attribute)*)? CLOSE_BRACE;

attribute
    : valueAttribute
    | constraintAttribute;

valueAttribute: key value?;

key: id;
value: BOOLEAN | FLOAT | INTEGER | STRING | attributes | vector;
vector: OPEN_BRACK (value (COMMA value)*)? CLOSE_BRACK;

constraintAttribute
    : 'constraint' constraint               # SingleConstraintAttribute
    | 'constraints' constraintList          # ListConstraintAttribute
    ;
constraintList: OPEN_BRACK (constraint (COMMA constraint)*)? CLOSE_BRACK;

constraints: 'constraints' NEWLINE INDENT constraintLine* DEDENT;

constraintLine: constraint NEWLINE;

constraint
    : equation                              # EquationConstraint
    | reference                             # LiteralConstraint
    | OPEN_PAREN constraint CLOSE_PAREN     # ParenthesisConstraint
    | NOT constraint                        # NotConstraint
    | constraint AND constraint             # AndConstraint
    | constraint OR constraint              # OrConstraint
    | constraint IMPLICATION constraint     # ImplicationConstraint
    | constraint EQUIVALENCE constraint     # EquivalenceConstraint
        ;

equation
    : expression EQUAL expression           # EqualEquation
    | expression LOWER expression           # LowerEquation
    | expression GREATER expression         # GreaterEquation
    | expression LOWER_EQUALS expression    # LowerEqualsEquation
    | expression GREATER_EQUALS expression  # GreaterEqualsEquation
    | expression NOT_EQUALS expression      # NotEqualsEquation
    ;

expression:
    FLOAT                                   # FloatLiteralExpression
    | INTEGER                               # IntegerLiteralExpression
    | STRING                                # StringLiteralExpression
    | aggregateFunction                     # AggregateFunctionExpression
    | reference                             # LiteralExpression
    | OPEN_PAREN expression CLOSE_PAREN     # BracketExpression
    | expression ADD expression             # AddExpression
    | expression SUB expression             # SubExpression
    | expression MUL expression             # MulExpression
    | expression DIV expression             # DivExpression
    ;
```

Tool	Graphical Editing	Textual Editing	Config-uration	Analysis	Trans-formation	Supported Lang. Levels
FeatureIDE [61]	✓	✗	✓	✓	✗	*Boolean*
flamapy [38]	✗	✗	✓	✓	✓	*Boolean*
Nemo [67]	✗	✗	✗	✓	✓	*Boolean*
TRAVART [34]	✗	✗	✗	✗	✓	*All*
UVLS [57]	✗	✓	✓	✓	✗	*All*
variability.dev [43]	✓	✗	✓	✓	✗	*Boolean*
ddueruem [42]	✗	✗	✗	✓	✗	*Boolean*
FM Fact Level [44]	✗	✗	✗	✓	✗	*Boolean*
pure::variants [71]	✓	✗	✓	✓	✗	*Boolean*
V4rdiac [28]	✗	✗	✓	✗	✗	*All*

Table 3: Tools integrating UVL either directly (upper part) or indirectly (lower part).

of optional and mandatory features themselves, or a single alternative, or an or group. However, FeatureIDE also does not support feature and group cardinalities. Constraints are limited to propositional logic constraints. Feature models created with FeatureIDE are usually serialised using an XML format. However, the serialisation can be changed to the UVL format [84], using the UVLFeatureModelFormat class.

The web-based feature-modelling tool variability.dev [43][8] builds on the ddueruem [42] analysis wrapper and the FeatureIDE [61] library. Thus, the expressiveness of the created UVL models is the same as those created with FeatureIDE and limited to the core of the UVL *Boolean*-level. However, using variability.dev, users have a low entry point for experimenting with feature modelling as users do not have to install a complete Eclipse-based application. Additionally, variability.dev allows collaborative editing of feature models. Created feature models can be downloaded either as a graphical image (SVG) or as a FeatureIDE XML file.

6.2.2. Textual editing

For textual editing of UVL models, Loth et al. [57] implemented the language server protocol for UVL in the tool UVLS. The language server protocol enables important language features, such as syntax highlighting, via a standardised interface. Thus, it can be integrated into common development environments, e.g., Visual Studio Code.[9] The current implementation of the language server protocol supports all language levels of UVL and features several analysis techniques [57] to enhance the textual editing of UVL models (cf. Section 6.2.3). For instance, UVLS checks whether the created UVL model is syntactically and semantically correct, e.g., avoiding void feature models. Furthermore, UVLS allows the configuration of a UVL model in a simplified configuration editor, similar to the one provided by FeatureIDE [61, 84].

[8] variability.dev – `https://variability.dev/`

[9] UVLS: `https://marketplace.visualstudio.com/items?itemName=caradhras.uvls-code`

6.2.3. Analysis

FeatureIDE [61, 84] can not only be used for graphical editing of feature models, but also for analysing them. Hence, FeatureIDE also enables the analysis of UVL models. However, the analysis is limited to the core of the *Boolean*-level, as this is the level supported by FeatureIDE (cf. Section 6.2.1).

The language server protocol implementation UVLS [57] provides syntactical and semantical analysis capabilities. For the syntactic check of a given UVL feature model, UVLS utilises the tree-sitter parser generator tool.[10] UVLS then checks if the tree-sitter parser accepts the given UVL model as a valid input. For the semantic analysis of a given UVL feature model, UVLS utilises the Z3 solver [23]. The SMT solver allows detecting if a UVL model does not allow valid configurations or contains any dead features or contradicting or redundant constraints.

flamapy [38] is a Python-based analysis framework for feature models.[11] The tool is plugin-based, utilising a core plugin orchestrating the execution of other plugins and also providing the hooks and frozen points of the framework [60]. Besides the core plugin, flamapy provides a feature model plugin, which supports the core of UVLs' *Boolean*-level and provides translations for PySAT, BDD support, and various input formats such as FeatureIDE [61] and S.P.L.O.T. [64]. Currently, flamapy supports multiple different solvers via the support of the PySAT4 metasolver[12], BDDs [40] and dependency graphs [59].

Nemo [67] allows counting valid configurations of numerical feature models via bit blasting [66]. The tool currently supports UVL as an input and output format (cf. Section 6.2.4). Therefore, Nemo utilises the *Boolean*-level of UVL, including group cardinalities. Based on the bit-blasted UVL model #SAT solvers and BDD solvers [40] are executed to count the number of valid configurations of the resulting model.

The online tool variability.dev [43] uses the analysis wrapper ddueruem [42] and the FeatureIDE [61] library to perform basic analysis on a created feature model. For instance, variability.dev detects dead features in a feature tree, or faulty configurations in the configuration editor (cf. Section 6.2.5).

6.2.4. Transformation

TraVarT [34] is a plugin-based variability model transformation environment.[13] At its core, TraVarT uses UVL as the pivot model. As the tool builds on the current Java implementation, TraVarT supports all language levels of UVL. Each plugin implements transformations between one variability artefact type and the UVL. These transformations are usually built by mapping core concepts of the supported variability model type onto the core concepts of UVL and vice versa [31, 32, 33]. For instance, in the available plugin for the DOPLER [25] decision modelling approach, a decision is mapped to a feature in the UVL. Also, a rule in the DOPLER decision model is mapped to either a

[10] tree-sitter: `https://tree-sitter.github.io/tree-sitter/`
[11] flamapy: `https://flamapy.github.io/`
[12] PySAT: `https://pysathq.github.io/`
[13] TraVarT: `https://github.com/SECPS/TraVarT`

feature property (mandatory), the feature model tree, or a constraint [33]. In the opposite direction, the hierarchy of the UVL feature model tree is captured via the visibility conditions of the DOPLER decision model.

Nemo [67] translates numerical feature models into UVL feature models using bit blasting [66]. Therefore, the bit-blasted numerical feature model is captured either as a DIMACS file, from which a UVL model is created or directly as a UVL model. Using Nemo the created UVL model can then be analysed (cf. Section 6.2.3).

6.2.5. Configuration

FeatureIDE [61, 84] also supports the configuration of feature models. Hence, FeatureIDE also enables the configuration of UVL models which support the core of the *Boolean*-level.

flamapy [38][14] utilises the configuration of UVL models, which support the core of the *Boolean*-level. flamapy uses the capability to validate if a configuration is valid for the given UVL model or to count valid configurations via state-of-the-art SAT solvers.

UVLS [57] supports configuring a given UVL model using a dedicated editor. The configuration editor supports the configuration of UVL models of all language levels. Therefore, UVLS presents a decision for each feature and its feature attributes to configure a configuration. The editor indicates if the given values for these features and their attributes still provide a basis for a valid configuration for the UVL model.

The online tool variability.dev [43] allows configuring created feature models using its configuration editor. The configuration editor supports the configuration of UVL models using the core of the *Boolean*-level. By default, the editor ensures that the selected configuration is valid, but also allows the configuration of invalid configurations. Configurations can be downloaded as a FeatureIDE configuration.

6.2.6. Others

UVL has also been either directly or indirectly, i.e., via one of the tools mentioned above, integrated into other tools. For instance, pure::variants [71] or ddueruem [42] support UVL via import and export capabilities [76]. Similarly, FM Fact Label [44] facilitates the visualisation of feature model metrics and supports common feature model formats, such as FeatureIDE [61], S.P.L.O.T. [64] or UVL. Other tools, such as V4rdiac [28], integrated TRAVART [34] to achieve variability model interoperability via the UVL or to facilitate the configuration of Cyber-Physical Production Systems [63]. The `UVLGenerator` can be used to generate UVL models whose structural properties can be customized according to the user's requirements [86]. Last but not least, UVLHub an open repository with UVL datasets is available [79][15].

[14] `https://www.flamapy.org/`
[15] `https://www.uvlhub.io/`

7. Discussion, Open Challenges and Future Work

To increase the adoption of UVL, its acceptance in industry is essential. To achieve that, we need to address the challenges that industry is having regarding variability modelling.

In 2020, Berger et al. provided some updates on industry challenges in SPLE [16] elicited earlier. At the SPLC 2023 Industry Challenges Workshop [10], 9 companies presented their challenges regarding variability management and systems and software product lines and discussed research opportunities. Addressing the challenges elicited in these recent works is essential for UVL to ensure adoption by industry. Of the many challenges discussed, especially the need to support multi product lines and system of systems product lines, efficient PL verification and validation, and tool support for integrated variability management across disciplines are relevant for the further development of UVL. The already available tooling and the extensibility of UVL should already help to address these challenges, however, further work needs to be done.

A recent paper [73] described specific challenges for UVL industry adoption, which we include:

- work with industry to empirically validate UVL and demonstrate it actually works for realistic cases. Extend or adapt UVL if necessary. Create demonstrators.
- develop extended tool support for modelling and configuration including generators for domain-specific artefacts.
- bridge the gap between UVL models and web-based (sales) configurators (see initial work by Abbasi et al. [1]).
- develop flexible mapping concept to support mapping of UVL features to solution space artefacts.
- develop consistency checking support intra- and inter-UVL models as well as between UVL models and artefacts.
- support the verification of UVL models and configurations
- support the automated creation of UVL models based on analysing existing variability information and existing artefacts [49] to extract variability.
- integrate UVL with tools used in industry such as ALM/PLM tools.
- support product line maintenance and evolution, e.g., develop automated refactoring support and a proper versioning concept and integrations with version management frameworks.
- work on the scalability of UVL to real-world systems. The multi-modelling concept of UVL thus should only be seen as a first step in this direction.
- investigate different visualisations of UVL models and configurations.

- provide material to train users as well as the advertise UVL.
- define UVL design patterns and guidelines [68].

Some of the challenges are being addressed and there are some solutions available. In future work, we plan to work on some of these challenges and also discuss them further with industry based on first case studies and demonstrators. We envision that this paper can also foster the community to investigate these challenges with further studies.

Besides industry adoption, we also plan to increase the adoption within the software product line and variability modelling community and beyond to the general software engineering community. Visibility at the main events as well as demonstrators and examples, together with guidance material, are essential to achieve that. Including further researchers in the MODEVAR initiative can have a snowball effect, if the researchers also start to use UVL in their collaborations with academia and industry as well as for teaching.

A key challenge to address is a more in-depth evaluation of UVL's simplicity, efficiency, and applicability in real-world scenarios. We show in this paper some indicators about these aspects, but a formal evaluation is still missing.

The area of teaching is yet another big opportunity to increase the adoption of UVL. The already existing documentation, example models, and UVL playground[16] are a very good starting point, however, we also need to prepare specific material for teaching UVL. MODEVAR community members needs to start using UVL in their teaching and report experiences.

8. Conclusions

During the last decades, feature modelling and analysis have been one of the main research topics in software product line engineering. UVL is a new language for textually modelling variability informed by a participatory process within the software product line community. The language is being used in different existing tools and is a proposal for the community to adopt in the future. A single language cannot fit all the variability needs of different scenarios, unless the language gets more and more complex to cover more needs. That is why UVL is designed using different language levels and includes extension mechanisms. As a result, UVL consists of a simple core language and allows users to extend the language to their specific needs. UVL then allows users to support all UVL models of other levels as well. Its simplicity allows information sharing among researchers, and we envision that it can be used in other scenarios, not only in software product lines.

Although the presented version of the language is stable, and we envision no major changes in the future, if UVL is widely adopted as we pursue, many challenges and research opportunities will appear. We plan to maintain and eventually enlarge a consortium of researchers who discuss the progress of the

[16] `https://universal-variability-language.github.io/`

language and agree on the language's evolution every year. We plan to enlarge and maintain the tool chain supporting UVL such as modelling [43, 84], analysis [38] or sharing [77] capabilities. With UVL, variability modelling can be adopted in many application domains and can be a central point for information sharing, tools integration, and variability modelling learning.

Material

All the source code and data can be downloaded and executed from the following repository: `https://github.com/Universal-Variability-Language`

Acknowledgements

Partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1608 – 501798263. It was also partially supported by FEDER/Ministry of Science, Innovation and Universities/Junta de Andalucía/State Research Agency/CDTI with the following grants: *Data-pl*(PID2022-138486OB-I00), TASOVA PLUS research network (RED2022-134337-T) and MIDAS (IDI-20230256). The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and the Christian Doppler Research Association is gratefully acknowledged. The authors thank Sebastian Krieter for insightful discussions on the language design of UVL and his contributions to the tools presented. Special thanks go to David Romero, who implemented UVLHub as a tool on top of UVL and helped us to detect some issues in the language. In addition, we would like to give a big thanks to all researchers who have contributed to the MODEVAR initiative throughout these years.

References

[1] Ebrahim Khalil Abbasi, Arnaud Hubaux, Mathieu Acher, Quentin Boucher, and Patrick Heymans. 2013. The anatomy of a sales configurator: An empirical study of 111 cases. In *Proceedings of the 25th International Conference on Advanced Information Systems Engineering*. Springer, 162–177.

[2] Mathieu Acher, Philippe Collet, David Benavides, and Rick Rabiser. 2020. Third International Workshop on Languages for Modelling Variability (MODEVAR@ SPLC 2020). In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*. 1–1.

[3] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2009. Composing Feature Models. In *Proc. Int'l Conf. on Software Language Engineering (SLE)*. Springer, Denver, CO, USA, 62–81.

[4] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. Familiar: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming (SCP)* 78, 6 (2013), 657–681.

[5] Prankur Agarwal, Kevin Feichtinger, Klaus Schmid, Holger Eichelberger, and Rick Rabiser. 2024. On the Challenges of Transforming UVL to IVML. `https://doi.org/10.48550/arXiv.2403.01952` arXiv:2403.01952 [cs].

[6] Mauricio Alférez, Mathieu Acher, José Angel Galindo, Benoit Baudry, and David Benavides. 2019. Modeling variability in the video domain: language and experience report. *Softw. Qual. J.* 27, 1 (2019), 307–347. `https://doi.org/10.1007/s11219-017-9400-8`

[7] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.* Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-37521-7

[8] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, Berlin, Heidelberg, 7–20. https://doi.org/10.1007/11554844_3

[9] Don Batory. 2020. *Automated Software Design Volume 1.* Lulu Press.

[10] Martin Becker, Rick Rabiser, and Goetz Botterweck. 2024. Not Quite There Yet: Remaining Challenges in Systems and Software Product Line Engineering as Perceived by Industry Practitioners. In *Proceedings of the 28th ACM International Systems and Software Product Line Conference.* ACM.

[11] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. 2019. Textual Variability Modeling Languages: An Overview and Considerations. In *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)* (Paris, France). ACM, New York, NY, USA, 151–157. https://doi.org/10.1145/3307630.3342398

[12] David Benavides, Rick Rabiser, Don Batory, and Mathieu Acher. 2019. First International Workshop on Languages for Modelling Variability (MODEVAR 2019). In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. 323–323.

[13] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–708.

[14] Thorsten Berger and Philippe Collet. 2019. Usage Scenarios for a Common Feature Modeling Language. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, 174–181.

[15] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)* (Pisa, Italy). ACM, New York, NY, USA, 7:1–7:8. https://doi.org/10.1145/2430502.2430513

[16] Thorsten Berger, Jan-Philipp Steghöfer, Tewfik Ziadi, Jacques Robin, and Jabier Martinez. 2020. The state of adoption and the challenges of systematic variability management in industry. *Empir. Softw. Eng.* 25, 3 (2020), 1755–1797.

[17] Mikaela Cashman, Justin Firestone, Myra B. Cohen, Thammasak Thianniwet, and Wei Niu. 2019. DNA as Features: Organic Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Paris, France). ACM, New York, NY, USA, 108–118. https://doi.org/10.1145/3336294.3336298

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

[19] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing Cardinality-Based Feature Models and Their Specialization. *Software Process: Improvement and Practice* 10 (2005), 7–29.

[20] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice* 10, 2 (2005), 143–169.

[21] Krzysztof Czarnecki and Chang Hwan Peter Kim. 2005. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *Proc. Int'l Workshop on Software Factories (SF)*. 16–20.

[22] Krzysztof Czarnecki and Andrzej Wasowski. 2007. Feature Diagrams and Logics: There and Back Again. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, Washington, DC, USA, 23–34.

[23] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[24] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Comm. ACM* 54, 9 (2011), 69–77.

[25] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. 2011. The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: A Multiple Case Study. *Automated Software Engineering* 18, 1 (2011), 77–114.

[26] Holger Eichelberger and Klaus Schmid. 2015. Mapping the Design Space of Textual Variability Modeling Languages: A Refined Analysis. *Int'l J. Software Tools for Technology Transfer (STTT)* 17, 5 (2015), 559–584. `https://doi.org/10.1007/s10009-014-0362-x`

[27] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the KConfig Semantics and its Analysis Tools. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)* (Pittsburgh, PA, USA). ACM, New York, NY, USA, 45–54. `https://doi.org/10.1145/2814204.2814222`

[28] Hafiyyan Sayyid Fadhlillah, Kevin Feichtinger, Philipp Bauer, Elene Kutsia, and Rick Rabiser. 2022. V4rdiac: tooling for multidisciplinary delta-oriented variability management in cyber-physical production systems. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B* (Graz, Austria) *(SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 34–37. `https://doi.org/10.1145/3503229.3547028`

[29] Hafiyyan Sayyid Fadhlillah and Rick Rabiser. 2024. Towards a Product Configuration Representation for the Universal Variability Language. In *Proceedings of the 28th ACM International Systems and Software Product Line Conference* (Dommeldange, Luxembourg) *(SPLC '24)*. Association for Computing Machinery, New York, NY, USA, 50–54. `https://doi.org/10.1145/3646548.3676544`

[30] Kevin Feichtinger and Jessie Galasso-Carbonnel. 2024. Seventh International Workshop on Languages for Modelling Variability (MODEVAR@SPLC 2024). In *Proceedings of the 28th ACM International Systems and Software Product Line Conference* (Dommeldange, Luxembourg) *(SPLC '24)*. Association for Computing Machinery, New York, NY, USA, 224. `https://doi.org/10.1145/3646548.3677006`

[31] Kevin Feichtinger, Kristof Meixner, Stefan Biffl, and Rick Rabiser. 2022. Evolution Support for Custom Variability Artifacts Using Feature Models: A Study in the Cyber-Physical Production Systems Domain. In *Reuse and Software Quality*, Gilles Perrouin, Naouel Moha, and Abdelhak-Djamel Seriai (Eds.). Springer International Publishing, Cham, 79–84.

[32] Kevin Feichtinger and Rick Rabiser. 2020. Towards Transforming Variability Models: Usage Scenarios, Required Capabilities and Challenges. In *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)* (Montreal, QC, Canada) *(SPLC '20)*. ACM, New York, NY, USA, 44–51. `https://doi.org/10.1145/3382026.3425768`

[33] Kevin Feichtinger and Rick Rabiser. 2020. Variability Model Transformations: Towards Unifying Variability Modeling. In *46th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, Portoroz, Slovenia.

[34] Kevin Feichtinger, Johann Stöbich, Dario Romano, and Rick Rabiser. 2021. TRAVART: An Approach for Transforming Variability Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)* (Krems, Austria). ACM, New York, NY, USA, Article 8, 10 pages. `https://doi.org/10.1145/3442391.3442400`

[35] Alexander Felfernig, Andreas Falkner, and David Benavides. 2024. *Feature Models: AI-Driven Design, Analysis and Applications*. Springer Nature. `https://doi.org/10.1007/978-3-031-61874-1`

[36] José A Galindo and David Benavides. 2020. A Python framework for the automated analysis of feature models: A first step to integrate community efforts. In *Proceedings of the 24th acm international systems and software product line conference-volume b*. 52–55.

[37] José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated Analysis of Feature Models: Quo Vadis? *Computing* 101, 5 (May 2019), 387–433. `https://doi.org/10.1007/s00607-018-0646-1`

[38] José A. Galindo, José Miguel Horcas, Alexander Felfernig, David Fernández-Amorós, and David Benavides. 2023. FLAMA: A Collaborative Effort to Build a New Framework for the Automated Analysis of Feature Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, 16–19. `https://doi.org/10.1145/3579028.3609008`

[39] Øystein Haugen, Andrzej Wasowski, and Krzysztof Czarnecki. 2012. CVL: Common Variability Language. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Salvador, Brazil). ACM, New York, NY, USA, 266–267. `https://doi.org/10.1145/2364412.2364462`

[40] Ruben Heradio, David Fernández-Amorós, José A. Galindo, David Benavides, and Don S. Batory. 2022. Uniform and Scalable Sampling of Highly Configurable Systems. *Empirical Software Engineering (EMSE)* 27, 2 (2022), 44. `https://doi.org/10.1007/s10664-021-10102-5`

[41] Tobias Heß, Tobias Müller, Chico Sundermann, and Thomas Thüm. 2022. ddueruem: A Wrapper for Feature-Model Analysis Tools. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Graz, Austria). ACM, New York, NY, USA, 54–57. `https://doi.org/10.1145/3503229.3547032`

[42] Tobias Heß, Tobias Müller, Chico Sundermann, and Thomas Thüm. 2022. ddueruem: a wrapper for feature-model analysis tools. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B* (Graz, Austria) *(SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 54–57. `https://doi.org/10.1145/3503229.3547032`

[43] Tobias Heß, Lukas Ostheimer, Tobias Betz, Simon Karrer, Tim Jannik Schmidt, Pierre Coquet, Sean Semmler, and Thomas Thüm. 2024. variability.dev: Towards an Online Toolbox for Feature Modeling. In *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)* (Bern, Switzerland). To appear.

[44] Jose M. Horcas, Jose A. Galindo, Mónica Pinto, Lidia Fuentes, and David Benavides. 2022. FM fact label: a configurable and interactive visualization of feature model characterizations. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B* (Graz, Austria) *(SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 42–45. `https://doi.org/10.1145/3503229.3547025`

[45] Jose-Miguel Horcas, Angela Villota, David Benavides, and Philippe Collet. 2022. Fifth International Workshop on Languages for Modelling Variability (MODEVAR@ SPLC 2022). In *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume A*. 264–264.

[46] ISO/IEC 26558:2017(en) 2017. *Software and systems engineering — Methods and tools for variability modelling in software and systems product line.* Standard. International Organization for Standardization/International Electrotechnical Commission, Geneva, CH. https://www.iso.org/obp/ui/#iso:std:iso-iec:26558:ed-1:v1:en

[47] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. 2008. Choco: An Open Source Java Constraint Programming Library. In *Proc. Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP)*. CCSD-HAL, Lyon, France.

[48] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-21. Software Engineering Institute, Pittsburgh, PA, USA.

[49] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. 2013. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Transactions on Software Engineering* 40, 1 (2013), 67–82.

[50] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A Tool Framework for Feature-Oriented Software Development. In *Proc. Int'l Conf. on Software Engineering (ICSE)* (Vancouver, Canada). IEEE, Washington, DC, USA, 611–614. https://doi.org/10.1109/ICSE.2009.5070568 Formal demonstration paper.

[51] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. 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)* (Paderborn, Germany). ACM, New York, NY, USA, 291–302. https://doi.org/10.1145/3106237.3106252

[52] Sebastian Krieter, Kevin Feichtinger, José A. Galindo, David Benavides, Rick Rabiser, Chico Sundermann, and Thomas Thüm. 2023. Second Tutorial on the Universal Variability Language. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Tokyo, Japan). ACM, New York, NY, USA, 273. https://doi.org/10.1145/3579027.3609002

[53] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. 2010. Model Counting in Product Configuration. In *Proc. Int'l Workshop on Logics for Component Configuration (LoCoCo)* (Edinburgh, UK). Open Publishing Association, Waterloo, Australia, 44–53. https://doi.org/10.4204/EPTCS.29.5

[54] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)* (Rochester, MI, USA). ACM, New York, NY, USA, 110:1–110:13. https://doi.org/10.1145/3551349.3556938

[55] Viet-Man Le, Thi Ngoc Trang Tran, Martin Stettinger, Lisa Weißl, Alexander Felfernig, Müslüm Atas, Seda Polat Erdeniz, and Andrei Popescu. 2021. Counteracting Exam Cheating by Leveraging Configuration and Recommendation Techniques. In *ConfWS*. 73–80.

[56] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. SAT-Based Analysis of Large Real-World Feature Models Is Easy. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, Berlin, Heidelberg, 91–100.

[57] Jacob Loth, Chico Sundermann, Tobias Schrull, Thilo Brugger, Felix Rieg, and Thomas Thüm. 2023. UVLS: A Language Server Protocol for UVL. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Tokyo, Japan). ACM, New York, NY, USA, 43–46. https://doi.org/10.1145/3579028.3609014

[58] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Evolution of the Linux Kernel Variability Model. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Jeju Island, South Korea). Springer, Berlin, Heidelberg, 136–150.

[59] Germán Márquez, José A. Galindo, Ángel Jesús Varela-Vaca, María Teresa Gómez López, and David Benavides. 2022. Advisory: vulnerability analysis in software development project dependencies. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B* (Graz, Austria) *(SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 99–102. https://doi.org/10.1145/3503229.3547058

[60] Simone Nasser Matos and Clovis Torres Fernandes. 2006. Early Definition of Frozen and Hot Spots in the Development of Domain Frameworks. In *Fourteenth ACM SIGSOFT Symposium on Foundations of Software Engineering.*

[61] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE.* Springer. https://doi.org/10.1007/978-3-319-61443-4

[62] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability With FeatureIDE.* Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-319-61443-4

[63] Kristof Meixner, Kevin Feichtinger, Hafiyyan Sayyid Fadhlillah, Sandra Greiner, Hannes Marcher, Rick Rabiser, and Stefan Biffl. 2024. Variability modeling of products, processes, and resources in cyber–physical production systems engineering. *Journal of Systems and Software* 211 (2024), 112007. https://doi.org/10.1016/j.jss.2024.112007

[64] Marcílio Mendonça, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, New York, NY, USA, 761–762.

[65] Marcílio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. 2009. SAT-Based Analysis of Feature Models Is Easy. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (San Francisco, California). Software Engineering Institute, Pittsburgh, PA, USA, 231–240.

[66] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Paris, France). ACM, New York, NY, USA, 289–301. https://doi.org/10.1145/3336294.3336297

[67] Daniel-Jesus Munoz, Mónica Pinto, Lidia Fuentes, and Don Batory. 2023. Transforming Numerical Feature Models into Propositional Formulas and the Universal Variability Language. *Journal of Systems and Software* 204 (2023), 111770. https://doi.org/10.1016/j.jss.2023.111770

[68] Damir Nesić, Jacob Krüger, Ștefan Stănciulescu, and Thorsten Berger. 2019. Principles of feature modeling. In *Proceedings of the 2019 27th ACM Joint Meeting of European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ACM, 62–73.

[69] Jeho Oh, Don Batory, and Rubén Heradio. 2023. Finding near-optimal configurations in colossal spaces with statistical guarantees. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023), 1–36.

[70] Terence Parr. 2013. *The definitive ANTLR 4 reference*. The Pragmatic Bookshelf. 1–326 pages.

[71] pure::systems. 2017. pure::variants. Website: `http://www.pure-systems.com/products/pure-variants-9.html`. Accessed: 2017-05-10.

[72] Rick Rabiser. 2019. Feature Modeling vs. Decision Modeling: History, Comparison and Perspectives. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 134–136. `https://doi.org/10.1145/3307630.3342399`

[73] Rick Rabiser. 2024. Industry Adoption of UVL: What We Will Need. In *Proceedings of the 28th ACM International Systems and Software Product Line Conference vol. 2*. ACM.

[74] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. 2002. Extending Feature Diagrams With UML Multiplicities. In *Proc. World Conf. on Integrated Design and Process Technology (IDPT)*.

[75] Jorge Rodas-Silva, José A Galindo, Jorge García-Gutiérrez, and David Benavides. 2019. Selection of software product line implementation components using recommender systems: An application to wordpress. *IEEE Access* 7 (2019), 69226–69245.

[76] Dario Romano, Kevin Feichtinger, Danilo Beuche, Uwe Ryssel, and Rick Rabiser. 2022. Bridging the gap between academia and industry: transforming the universal variability language to pure::variants and back. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B* (Graz, Austria) *(SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 123–131. `https://doi.org/10.1145/3503229.3547056`

[77] David Romero, José A. Galindo, José Miguel Horcas, and David Benavides. 2021. A First Prototype of a New Repository for Feature Model Exchange and Knowledge Sharing. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, 80–85. `https://doi.org/10.1145/3461002.3473949`

[78] David Romero-Organvidez, Jose A. Galindo, and David Benavides. 2024. UVL Sentinel: a tool for parsing and syntactic correction of UVL datasets. arXiv:2403.18482 [cs.SE] `https://arxiv.org/abs/2403.18482`

[79] David Romero-Organvidez, José A. Galindo, Chico Sundermann, Jose-Miguel Horcas, and David Benavides. 2024. UVLHub: A feature model data repository using UVL and open science principles. *Journal of Systems and Software* (2024), 112150. `https://doi.org/10.1016/j.jss.2024.112150`

[80] Valentin Rothberg, Nicolas Dintzner, Andreas Ziegler, and Daniel Lohmann. 2016. Feature Models in Linux: From Symbols to Semantics. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)* (Salvador, Brazil). ACM, New York, NY, USA, 65–72. `https://doi.org/10.1145/2866614.2866624`

[81] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic Semantics of Feature Diagrams. *Computer Networks* 51, 2 (2007), 456–479.

[82] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. 2021. Yet Another Textual Variability Language? A Community Effort Towards a Unified Language. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Leicester, UK). ACM, New York, NY, USA, 136–147. `https://doi.org/10.1145/3461001.3471145`

[83] Chico Sundermann, Kevin Feichtinger, José A. Galindo, David Benavides, Rick Rabiser, Sebastian Krieter, and Thomas Thüm. 2022. Tutorial on the Universal Variability Language. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Graz, Austria). ACM, New York, NY, USA, 260:1. `https://doi.org/10.1145/3546932.3547024`

[84] Chico Sundermann, Tobias Heß, Dominik Engelhardt, Rahel Arens, Johannes Herschel, Kevin Jedelhauser, Benedikt Jutz, Sebastian Krieter, and Ina Schaefer. 2021. Integration of UVL in FeatureIDE. In *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)* (Leicester, UK). ACM, New York, NY, USA, 73–79. `https://doi.org/10.1145/3461002.3473940`

[85] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2023. Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. *Empirical Software Engineering (EMSE)* 28, 29 (Jan. 2023), 38. `https://doi.org/10.1007/s10664-022-10265-9`

[86] Chico Sundermann, Tobias Heß, Rahel Sundermann, Elias Kuiter, Sebastian Krieter, and Thomas Thüm. 2024. Generating Feature Models with UVL's Full Expressiveness. In *Proceedings of the 28th ACM International Systems and Software Product Line Conference* (Dommeldange, Luxembourg) *(SPLC '24)*. Association for Computing Machinery, New York, NY, USA, 61–65. `https://doi.org/10.1145/3646548.3676602`

[87] Chico Sundermann, Stefan Vill, Thomas Thüm, Kevin Feichtinger, Prankur Agarwal, Rick Rabiser, José A. Galindo, and David Benavides. 2023. UVLParser: Extending UVL With Language Levels and Conversion Strategies. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Tokyo, Japan). ACM, New York, NY, USA, 39–42. `https://doi.org/10.1145/3579028.3609013`

[88] Thomas Thüm. 2020. A BDD for Linux? The Knowledge Compilation Challenge for Variability. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Montreal, QC, Canada). ACM, New York, NY, USA, Article 16, 6 pages. `https://doi.org/10.1145/3382025.3414943`

[89] Thomas Thüm, Philippe Collet, and Mathieu Acher (Eds.). 2021. *Fourth International Workshop on Languages for Modelling Variability (MODEVAR@SPLC 2021)* (Leicester, UK). ACM, New York, NY, USA. `https://doi.org/10.1145/3461001.3473056`

[90] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Munich, Germany). IEEE, Washington, DC, USA, 191–200. `https://doi.org/10.1109/SPLC.2011.53`

[91] Thomas Thüm, Christoph Seidl, and Ina Schaefer. 2019. On Language Levels for Feature Modeling Notations. In *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)* (Paris, France). ACM, New York, NY, USA, 158–161. `https://doi.org/10.1145/3307630.3342404`

[92] Andrzej Wasowski and Thorsten Berger. 2023. *Domain-Specific Languages: Effective modeling, automation, and reuse.* Springer.

[93] Wei Zhang, Haiyan Zhao, and Hong Mei. 2004. A Propositional Logic-Based Method for Verification of Feature Models. In *Proc. Int'l Conf. on Formal Engineering Methods (ICFEM)*. Springer, Berlin, Heidelberg, 115–130.

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: