On Language Levels for Feature Modeling Notations

Thomas Thüm
TU Braunschweig
Brunswick, Germany

Christoph Seidl
IT University
Copenhagen, Denmark

Ina Schaefer
TU Braunschweig
Brunswick, Germany

ABSTRACT
Configuration is a key enabling technology for the engineering of systems and software as well as physical goods. A selection of configuration options (aka. features) is often enough to automatically generate a product tailored to the needs of a customer. It is common that not all combinations of features are possible in a given domain. Feature modeling is the de-facto standard for specifying features and their valid combinations. However, a pivotal hurdle for practitioners, researchers, and teachers in applying feature modeling is that there are hundreds of tools and languages available. While there have been first attempts to define a standard feature modeling language, they still struggle with finding an appropriate level of expressiveness. If the expressiveness is too high, the language will not be adopted, as it is too much effort to support all language constructs. If the expressiveness is too low, the language will not be adopted, as many interesting domains cannot be modeled in such a language. Towards a standard feature modeling notation, we propose the use of language levels with different expressiveness each and discuss criteria to be used to define such language levels. We aim to raise the awareness on the expressiveness and eventually contribute to a standard feature modeling notation.

CCS CONCEPTS
• Software and its engineering → Software product lines.

KEYWORDS
product lines, variability modeling, feature model, language design, expressiveness, automated analysis

ACM Reference Format:

1 MOTIVATION
In industrial software and systems engineering, there is an increasing awareness that reuse is central for quality and development efficiency [7, 8]. With software product lines, software-intensive systems are derived from reusable assets [44]. Ideally, software products are generated automatically for a given selection of features [3]. However, such configuration is not only used for software or software-intensive systems, but also for arbitrary products following the vision of mass customization [52]. For instance, in an industrial collaboration we modularized financial products, such as loans, into features for improved productivity and consistency compared to a product-by-product development.

While mass customization is applied to various domains, they all have in common that there are constraints among those features, meaning that not all combinations of features are valid. Constraints can have numerous sources [39] and are typically specified by means of feature modeling [4, 26]. Other related, but less prominent approaches to specify constraints are decision models [18, 24] and orthogonal variability models [44]. Since feature models have been introduced in 1990 [26], numerous feature modeling notations have been proposed [14, 25, 47], including graphical notations such as CVL [22] and textual notations such as TVL [12] and KConfig [19]. There are numerous tools for feature modeling available [35], such as FeatureIDE [34], SPlot [36], FAMA [6], and Betty [49].

The large number of feature modeling notations and languages is a pivotal hurdle for practitioners, researchers, and teachers. Practitioners need interoperability to combine different tools. Researchers need tool support to evaluate their research and reduced effort for tool building. Teachers want to give students hands-on experience with feature modeling, which typically involves teaching a number of notations and tools. Ideally, researchers, practitioners, teachers, and tool builders agree on a single feature modeling language, but this is unlikely to happen, as feature modeling is used for various, very different domains. A language that can express everything required for all domains is too much effort to integrate into tools. In contrast, a core language that is easy to support in tools is most likely not sufficient to express relevant industrial domains.

We aim to tackle this problem by proposing the use of language levels so that tool builders can decide which language levels to support. Those levels are designed to achieve a trade-off between the expressiveness of the modeling language and its applicability to different domains and the capabilities required by a respective tool. Our discussion is inspired by the Java Modeling Language (JML), which is a behavioral interface specification language comprising different language levels [31]. JML is used for many different applications and tools [10], where tool builders can decide which language levels they support [31].

2 MAJOR LEVELS FOR FEATURE MODELING
Our experience with applying feature modeling to industrial applications is that their most important capability is the automated reasoning about their constraints. For instance, in an industrial application where Excel was used to specify about 500 features and 2,000 rules among them, we found that 20% of the features were dead (i.e., were not contained in any valid configuration) and
about half of the rules were redundant (i.e., their removal would not change the set of valid configurations). In contrast, in another industrial project with about 700 features where FeatureIDE [34] was used, we found a number of redundant constraints but not a single dead feature. A likely reason for the absence of dead features is that FeatureIDE highlights dead features directly in the feature diagram. Even though these are just two cases, it indicates that humans are not able to reason about configuration spaces with hundreds of features and thousands of constraints.

Reasoning about feature models is typically performed by translating the feature model into a logical representation [4, 16, 37] and then encoding analyses as satisfiability problems [5, 21, 55]. Satisfiability problems have the advantage that they can be delegated to efficient off-the-shelf solvers, such as SAT solvers, SMT solvers, CSP solvers, or binary decision diagrams. It is even possible to combine several solvers [49] or to implement dedicated data structures being used in combination with state-of-the-art solvers [30, 49]. The encoding as satisfiability problems also has the advantage that explanations for unsatisfiable problems (e.g., dead features or redundant constraints) can be delegated to algorithms computing minimal unsatisfiable cores [2, 29, 41].

The automated analysis of feature models to detect anomalies, such as dead features or redundant constraints [5, 21], is just one of many applications where tools need to reason about the constraints of the feature model. Besides the analysis of the feature model in isolation, there are numerous different analyses that also incorporate other artifacts, such as source code [55]. Reasoning about feature models is used for parsing [27], dead-code analysis [53], code simplification [59], type checking [54], consistency checking [15], dataflow analyses [32], model checking [13], testing [11] including variability-aware execution [40] and sampling [33, 58], product configuration [46], optimization of non-functional properties [51], and variant-preserving refactoring [20].

The large number of applications that require to reason about constraints indicates the importance of solvers for feature modeling. In order to be able to reduce those problems to dedicated solvers, it is vital that the expressiveness of a feature modeling languages is not larger than that of given solvers. For example, SAT solvers can solve satisfiability problems in propositional logic, whereas SMT solvers support fragments of first-order logic [9]. Most feature modeling notations can easily be translated into propositional logic. However, extended feature models supporting numerical attributes or cardinality-based feature models with feature cardinalities cannot easily be encoded with propositional logic. Hence, there is a trade-off between high expressiveness in the feature modeling language and the number of solvers that are applicable to reason about constraints. Clearly, one could argue that we can do all analyses with SMT solvers only, but our experience is that SMT solvers are magnitudes slower than SAT solvers for many satisfiability problems. Consequently, a reduced expressiveness leads to more solvers being applicable and potentially also to faster solving times.

Based on prior discussions about differing expressiveness of SAT and SMT solvers, we propose two major language levels for feature modeling notations: Level 1 for those notations that can be encoded directly as SAT problem and Level 2 for those that can be encoded as SMT problem, but not as a SAT problem. However, it is not yet clear whether these two levels are sufficient. It requires a community effort to identify (a) solvers that are relevant to reason about feature models and (b) classes of those solvers with the same expressiveness. For instance, it is known that SAT solvers and binary decision diagrams can both be used to decide satisfiability of propositional formulas and both have often been applied to reason about feature models. As a consequence, we do not yet know how many levels are necessary, but at least two levels are required.

### 3 MINOR LEVELS FOR FEATURE MODELING

In the previous section, we argued for language levels aligned with the expressiveness of state-of-the-art solvers. We refer to those levels as major language levels, as those are likely to have the largest impact on the expressiveness of a feature modeling notation. However, these are clearly not the only requirements for feature modeling languages. This can be illustrated easily by considering the standard input language of SAT solvers, namely DIMACS. DIMACS is a standardized format supported by all SAT solvers, in which a propositional formula is specified in conjunctive normal form, and variables are named $1, 2, \ldots, n$. The success of the SAT community is likely also due to having agreed on this standard format, such that a standard format could have similar effects for feature modeling. However, the DIMACS format is not suitable for feature modeling, as feature models tend to explode in conjunctive normal form [28] and as the hierarchy is crucial for its success.

We argue that even for each major level it is not feasible to agree on a set of language constructs, which gives rise to minor language levels for each major level. While major levels are driven by solving, minor levels are driven by two requirements. First, one should take into account existing languages, as import and export to these languages is vital to the adoption for a new language. Ideally, tools would even use the standard language opposed to their own language in the long run. Second, the design of a language should be influenced by requirements of real-world domains. That is, the language needs to be driven at least to some extent by what is required to model real-world constraints. Nevertheless, it is not useful to consider every language construct of existing languages and every requirement from real-world domains.

The expressiveness of numerous early notations for feature modeling languages has been formalized already in 2007 [47]. This can be a starting point for considerations regarding language constructs for certain levels. If removing a language construct from a language does not reduce its expressiveness, it is considered syntactic sugar. Two levels could only differ in syntactic sugar to reduce the burden of tool developers to support too many language constructs. If two levels are not equally expressive, transformations are still possible between each other but they may result in a homomorphism (i.e., they lose information).

We give an example for a trade-off in terms of expressiveness. Some feature modeling notations allow arbitrary propositional formulas as cross-tree constraints (aka. complex constraints), while others only support requires and excludes constraints between a pair of features (aka. simple constraints) [28]. Having only simple constraints is much easier to handle for tools, but domain modelers may need to create numerous additional abstract features [56] to model real-world constraints [28]. In contrast, with complex constraints it is possible to create constraints which are hard to
translate into conjunctive normal form. For instance, to the best of our knowledge, the Linux kernel feature model cannot be translated into conjunctive normal form without introducing new variables. Tools that need to transform the Linux model into conjunctive normal form use the Tseitin transformation. For Linux, the Tseitin transformation results in about 45,000 new variables for a model with 15,000 variables in the original version [43]. Minor levels could be used to distinguish between notations that support complex constraints and others that only support simple constraints.

5 CONCLUSION AND FUTURE WORK
A standard feature modeling notation can have a great impact on the interoperability of tools. Interoperability may help for industrial adoption by practitioners, researchers to use tools in a plug-and-play manner, and teachers to avoid the introduction of numerous notations. We identified that it is unlikely that a single feature modeling notation will be sufficient for all domains and propose to make use of different language levels. First, major language levels are aligned with the expressiveness of solver classes. Second, minor language levels can be defined inside each major language level to address trade-offs for different purposes. Third, orthogonal language levels can be combined with any previously mentioned major or minor language level (e.g., to facilitate modularity or to model evolving configuration spaces). Which concrete language levels should be considered requires a community effort, which we hope to initiate with this article.

ACKNOWLEDGMENTS
We gratefully acknowledge discussions on language levels for automated analysis of feature models with Maurice H. ter Beek. This work has been partially supported by the German Research Foundation within the project VariantSync (TH 2387/1-1).

REFERENCES