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 wells 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:
Thomas Thüm, Christoph Seidl, and Ina Schaefer. 2019. On Language Levels for Feature Modeling Notations. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3307630.3342404

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

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342404

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 Tseytin transformation. For Linux, the Tseytin 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.

4 ORTHOGONAL LEVELS

While minor language levels are concrete instantiations of a major language level, there are also language constructs which do not influence the expressiveness of the modeling notations. Hence, those language constructs are somewhat orthogonal to major and minor language levels and may be combined with any of them. We give two concrete examples for such orthogonal language constructs.

First, a common problem is that feature models tend to grow in size. In our experience, feature models with more than 100 features are hard to understand for domain modelers. Furthermore, any change in the model can potentially introduce anomalies, such as dead features, in any other part of the model as constraints are globally visible. A common method in software engineering to achieve modularity is to apply information hiding. For feature models, feature-model interfaces hide some features and some constraints when feature models are composed with other feature models [48, 57]. Numerous techniques have been proposed to compose feature models [17, 23, 38, 45], which could be combined with feature-model interfaces to achieve modularity and, thus, to support the modeling of large configuration spaces in manageable modules. Even if feature-model interfaces are not available in the language, modular reasoning is feasible by means of slicing [1] and the computation of implicit constraints [2]. Such a modularity is possible at every language level. It is an open question whether modularization concepts should be part of the language that has to be supported by all tools or whether it is an optional and then orthogonal language construct for each language level.

Second, there are several existing feature modeling languages that allow to model the evolution of configuration spaces explicitly. Hyper-feature models [50] allow to refer to features in numerous versions, and even constraints can be defined for certain versions only. Temporal feature models [41] allow to capture every possible evolution scenario explicitly. Both, hyper-feature models and temporal feature models support different aspects of evolving configuration spaces and can be added on top of any feature modeling notation. It is a point for discussion whether this needs to be supported by every tool, as the evolution of feature models may also be tracked by version control instead, and new versions of features can also be supported by newly introduced features. However, in industrial projects, we experienced the need for dedicated languages constructs for evolving configuration spaces. A typical example is that domain modelers want to plan certain changes of the feature model for the future, while they still want to fix wrong constraints in the current version in an integrated manner [42].

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

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2011. Slicing Feature Models. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 424–427.
[2] Sofia Ananieva, Matthias Kowal, Thomas Thüm, and Ina Schaefer. 2016. Implicit Constraints in Partial Feature Models. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 18–27.
[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.
[4] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 7–20.
[5] 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.
[6] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2007. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. Technical Report 2007-01, Lero, 129–134.
[7] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 7:1–7:8.
[8] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 73–82.
[9] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. 2009. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands.
[10] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. 2005. An Overview of JML Tools and Applications. *Int'l J. Software Tools for Technology Transfer (STTT)* 7, 3 (2005), 212–232.
[11] Ivan Do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. 2014. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *J. Information and Software Technology (IST)* 56, 10 (2014), 1183–1199.
[12] 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.
[13] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. Software Engineering (TSE)* 39, 8 (2013), 1069–1089.

[14] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 173–182.

[15] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 211–220.

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

[17] Deepak Dhungana, Paul Grünbacher, Rick Rabiser, and Thomas Neumayer. 2010. Structuring the Modeling Space and Supporting Evolution in Software Product Line Engineering. *J. Systems and Software (JSS)* 83, 7 (2010), 1108–1122.

[18] Rebecca Duray, Peter T. Ward, Glenn W. Milligan, and William L. Berry. 2000. Approaches to Mass Customization: Configurations and Empirical Validation. *J. Operations Management (JOM)* 18, 6 (2000), 605–625.

[19] 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)*. ACM, 45–54.

[20] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. 2017. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 316–326.

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

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

[23] Gerald Holl, Paul Grünbacher, and Rick Rabiser. 2012. A Systematic Review and an Expert Survey on Capabilities Supporting Multi Product Lines. *J. Information and Software Technology (IST)* 54, 8 (2012), 828–852.

[24] Arnaud Hubaux, Dietmar Jannach, Conrad Drescher, Leonardo Murta, Tomi Männistö, Krzysztof Czarnecki, Patrick Heymans, Tien N. Nguyen, and Markus Zanker. 2012. Unifying Software and Product Configuration: A Research Roadmap. In *Proc. Configuration Workshop (ConfWS)*. 31–35.

[25] Arnaud Hubaux, Thein Than Tun, and Patrick Heymans. 2013. Separation of Concerns in Feature Diagram Languages: A Systematic Survey. *Comput. Surveys* 45, 4, Article 51 (2013), 51:1–51:23 pages.

[26] 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.

[27] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 805–824.

[28] 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)*. ACM, 291–302.

[29] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. 2016. Explaining Anomalies in Feature Models. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 132–143.

[30] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating Configuration Decisions with Modal Implication Graphs. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 898–909.

[31] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. 2013. *JML Reference Manual*.

[32] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 81–91.

[33] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 643–654.

[34] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[35] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, and Gunter Saake. 2014. An Overview on Analysis Tools for Software Product Lines. In *Proc. Workshop on Software Product Line Analysis Tools (SPLat)*. ACM, 94–101.

[36] 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, 761–762.

[37] Marcílio Mendonça, Andrzej Wąsowski, Krzysztof Czarnecki, and Donald Cowan. 2008. Efficient Compilation Techniques for Large Scale Feature Models. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 13–22.

[38] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. 2007. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *Proc. Int'l Conf. on Requirements Engineering (RE)*. IEEE, 243–253.

[39] Sarah Nadi, Thorsten Berger, Christian KÃdstner, and Krzysztof Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Trans. Software Engineering (TSE)* 41, 8 (2015), 820–841.

[40] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 907–918.

[41] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly Analyses for Feature-Model Evolution. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 188–201.

[42] Michael Nieke, Christoph Seidl, and Thomas Thüm. 2018. Back to the Future: Avoiding Paradoxes in Feature-Model Evolution. In *Proc. Int'l Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution)*. ACM, 48–51.

[43] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM. To appear.

[44] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[45] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. 2011. Multi-Dimensional Variability Modeling. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 11–22.

[46] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. 2013. Scalable Product Line Configuration: A Straw to Break the Camel's Back. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 465–474.

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

[48] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 667–678.

[49] Sergio Segura, José A. Galindo, David Benavides, José A. Parejo, and Antonio Ruiz-Cortés. 2012. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 63–71.

[50] Christoph Seidl, Ina Schaefer, and Uwe A. 2014. Capturing Variability in Space and Time with Hyper Feature Models. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 6, 6:1–6:8 pages.

[51] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. *Software Quality Journal (SQJ)* 20, 3-4 (2012), 487–517.

[52] Giovani Da Silveira, Denis Borenstein, and FlÃąvio S. Fogliatto. 2001. Mass Customization: Literature Review and Research Directions. *Int'l J. Production Economics* 72, 1 (2001), 1–13.

[53] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. Europ. Conf. on Computer Systems (EuroSys)*. ACM, 47–60.

[54] Sahil Thaker, Don Batory, David Kitchin, and William Cook. 2007. Safe Composition of Product Lines. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 95–104.

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

[56] 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)*. IEEE, 191–200.

[57] Thomas Thüm, Tim Winkelmann, Reimar Schröter, Martin Hentschel, and Stefan Krüger. 2016. Variability Hiding in Contracts for Dependent Software Product Lines. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 97–104.

[58] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 1–13.

[59] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 178–188.