A multi-level approach for supporting configurations: A new perspective on software product line engineering

CLARK, Tony <http://orcid.org/0000-0003-3167-0739>, FRANK, Ulrich, REINHARTZ-BERGER, Iris and STURM, Arnon

Published version

CLARK, Tony, FRANK, Ulrich, REINHARTZ-BERGER, Iris and STURM, Arnon (2017). A multi-level approach for supporting configurations: A new perspective on software product line engineering. In: CABANILLAS, Cristina, ESPAÑA, Sergio and FARSHIDI, Siamak, (eds.) Proceedings of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th International Conference on Conceptual Modelling (ER 2017), Valencia, Spain, - November 6-9, 2017. CEUR Workshop Proceedings (1979). CEUR-WS, 170-178.

A Multi-Level Approach for Supporting Configurations: A new Perspective on Software Product Line Engineering

Ulrich Frank[1], Iris Reinhartz-Berger[2], Arnon Sturm[3], Tony Clark[4]

[1] Universität Duisburg-Essen, Germany
[2] University of Haifa, Israel
[3] Ben-Gurion University of the Negev, Israel
[4] Sheffield-Hallam University, UK
ulrich.frank@uni-due.de, iris@is.haifa.ac.il, sturm@bgu.ac.il, t.clark@shu.ac.uk

Abstract. Configuration is a common way in many markets to cope with reducing costs and improving customer satisfaction. There are various approaches to represent product configurations, the most common of which is feature modeling. However, feature models suffer from principal limitations, including ambiguity and lack of abstraction, increasing maintainability effort and limiting lifecycle support. In this paper, we suggest using a multi-level modeling approach to improve flexibility, reuse, and integrity and demonstrate the advantages of the approach over feature modeling.

Keywords: multi-level modelling, domain specific modelling languages, configuration, variability, software product line engineering, feature modeling

1 Introduction

The variety and complexity of systems have dramatically increased in the last two decades. These introduce challenges to the development of software that the well-known modeling paradigms could not handle without modifications and adaptations. Software Product Line Engineering (SPLE) [10, 16] suggests reducing variety and complexity of development and management by handling product families rather than individual products and promoting systematic reuse across products. In this context, variability management plays an essential role. Variability is defined as the ability of a software system or a software artifact to be changed so that it fits a specific context [19]. A common way to realize variability is through configuration [4, 14]. Configuration enables choosing alternatives that may be specified either explicitly or implicitly. Particularly, potential configurations are all alternatives of an artifact, whereas valid configurations are all potential alternatives that satisfy given constraints.

Current approaches to SPLE adopt a two-layered framework [16]: domain engineering containing the specification and implementation of product line artifacts (potential configurations) and application engineering consisting the specification and implementation of specific product artifacts (valid configurations). The rigid division

into these two levels imposes limitations on the ability to abstract concepts that are used across these levels and to define the semantics of the various models which may be expressed in different languages. Moreover, checking models for consistency and correctness is challenging in the general context of SPLE and particularly with respect to variability management. Scalability and visualization/graphical overload are mentioned among the most prominent challenges [3].

To address the above challenges, we propose to adopt a Multi-Level Modeling (MLM) approach. Generally, MLM [1, 2, 11] is a new modeling paradigm that supports abstraction through the use of both inheritance and meta-types, the latter is used to allow the modeler to integrate the language used for modeling together with the models themselves. As such MLM has the potential to support semantics integration, abstraction and reuse, as well as to define Domain-Specific Modeling Languages (DSML).

In [18], we have explored MLM in the context of SPLE variability mechanisms – techniques applied in order to adapt software product line artifacts to the context of particular products. Here, we aim to analyze prospects of applying MLM to SPLE in the context of variability management. Particularly, we demonstrate the limitations of feature modeling [6], a well-known paradigm for variability management, which supports configuration specification. We further propose an MLM approach [12] to address those drawbacks. This approach enables the creation of DSMLs on different levels of abstraction, where a lower level DSML is specified by a higher level DSML. In such an architecture the traditional dichotomy between modeling language and model is relaxed: each DSML is specified through a model which in turn was created with a higher level language. The approach enables a common representation of models and code, which promotes elegant specification and implementation of software product lines.

The rest of the paper is structured as follows. Section 2 reviews current approaches to variability modeling, focusing on feature modeling and exemplifying its limitations. Section 3 introduces the MLM approach and demonstrates its potential to model configurations and to address feature modeling drawbacks. Finally, Section 4 discusses future research directions.

2 Current Approaches to Model Variability

Variability management has gained interest in the past decade [6]. Many of the approaches refer to modeling, utilizing feature modeling, UML extensions, formal (mathematical) techniques, and more. Of those, feature modeling is the mostly used paradigm, both in research [6] and in industry [5]. Commonality and variability specification is supported in feature modeling through the notion of features – prominent or distinctive user-visible aspects, qualities, or characteristics of a software system or systems. Dependencies among features enable constraining valid configurations through different utilities, such as mandatory and optional features and variants (using OR and XOR relations). Over the years, various extensions to the original feature models [15] have been suggested to address limitations in expressiveness. These extensions include among others adding cardinalities to features to enable specifications

other than mandatory and optional features, adding cardinalities to dependencies (groups of features) to enable specifications other than OR and XOR relations, and supporting the specification of value choice from large or infinite domains (attributes) [7]. Moreover, feature-oriented programming (FOP) [17] has been proposed in order to percolate feature modeling concepts into code and support software modularization based on composition mechanisms, called refinements. Yet, to the best of our knowledge, the transition from feature models to feature-oriented code is not fully automated.

While feature modeling may serve as a versatile tool for representing variability in requirements, they are limited in specifying and designing software product lines. In order to demonstrate the limitations, we use the feature model depicted in Fig. 1. The model specifies a family of software systems for bicycle dealerships. Each system needs to represent and manage domain objects – bicycles, which exist in a remarkable variety. Some dealers need to cover the full range in great detail (including the actual distinction between racing bikes and pro-racers). Others are more satisfied with a more generic concept of bicycle, which is characterized by its weight, its parts (fork, frame, two wheels), and so on.

Analyzing the bicycles model, we observe the following limitations.

Ambiguity: Apparently, a feature may represent a class (e.g., frame or fork), an attribute (e.g., size or weight), a possible attribute value (e.g., alum or carbon), or even the result of an operation (e.g., the weight of a bicycle is the sum of its constituents' weights). In an early stage of requirements analysis, it can be a good idea to use such an abstraction. However, the closer one gets to the design phase, the more problematic this abstraction may be: the concept of a feature does not allow for a clear correspondence to software design concepts such as class, attribute, etc. Therefore, feature models do not allow for a straightforward transformation to design documents such as object models, and the synchronization of design documents and feature models is a remarkable challenge that requires introduction of additional languages and tracing capabilities, such as in orthogonal variability modeling [16].

Lack of abstraction: The lack of classification will not only prevent modellers from expressing knowledge they have, it also creates a threat to maintainability and integrity. Since it is not possible to define classes that are characterized by a certain feature configuration, it is not possible to define constraints that apply to these classes. Defining corresponding constraints directly on features is not only cumbersome, it may in the end even be a wasted exercise, because it can create additional complexity that compromises a model's readability. Since feature models lack the concept of class, they do not support a well-founded concept of generalization/specialization either. Therefore, redundancy will often be unavoidable, which jeopardizes a model's integrity.

To demonstrate this limitation, consider the need to specify classes of racing bikes and pro-racers. Both are specializations of bicycles, whereas pro-racers are special racing bikes. The following constraints can be introduced for constraining valid configurations of those classes of bicycles:

1.	Racing => ¬Suspension ∧ ¬SaftyRelf ∧ ¬AllTerrain ∧ ¬AllTerrain1 ∧ Race ∧ Race1 ∧ City
2.	ProRacer => Racing ∧ ¬MudMount ∧ Tubless ∧ UClcretified ∧ Carbon1

This specification hinders the hierarchy of the three domains (regular bicycles, racing bikes, and pro racers). It makes the models less understandable for the modellers, especially when analyzing the consequences of feature modification, as all features appear in the same (domain engineering) level without separation into the different bicycles classes.

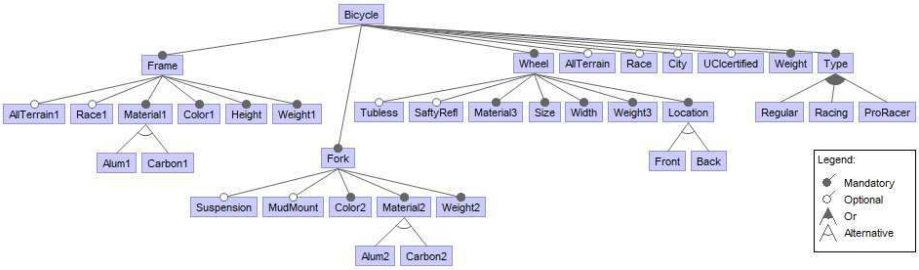

Fig. 1. A bicycle feature model

Another example of the lack of abstraction refers to the need to specify that the size of the front wheel equals to the size of the back wheel, but its weight is lighter. Here, we need the distinction between classes, attributes, and attribute values. The expressiveness of feature models is insufficient, and hence we have to use extensions that support specification of feature cardinalities and attributes, e.g., the extension proposed in [7]. The cardinality of the feature *wheel* should be specified as [2..2] (exactly 2) and the features *size* and *weight3* should be replaced by attributes (of type real). The constraints can be then specified as follows:

3. Wheel[1].Location.Front =>	Wheel[2].Location.Back ∧ Wheel[1].Size = Wheel[2].Size ∧ Wheel[1].Weight3 < Wheel[2].Weight3
4. Wheel[1].Location.Back =>	Wheel[2].Location.Front ∧ Wheel[1].Size = Wheel[2].Size ∧ Wheel[2].Weight3 < Wheel[1].Weight3

This specification is complicated and its maintainability and comprehensibility are further questioned. Hence, in the following we propose a multi-level modelling language for supporting variability modeling in software product lines without forcing overload of concepts.

3 Beyond Feature Models: Prospects of an MLM Approach

To develop a convincing rationale for this proposal we first give a brief overview of core concepts of MLM. Then, we outline why the additional abstraction enabled by MLM is promising for designing and managing product lines. Finally, we illustrate

the potential of MLM for overcoming the aforementioned limitations using the bicycles example.

3.1 Multi-Level Modelling in a Nutshell

In the traditional paradigm, all entity types or classes of a model are located on the same level of classification, which is usually M1. Therefore, it is not possible to express knowledge about classes of classes. Furthermore, classes cannot be modelled as objects that have a state and can execute operations. This lack of abstraction is sometimes handled by overloading one level of classification. However, the ambiguity caused by overloading creates a serious threat to integrity. A further option would be to extensionally model all cases that were otherwise covered by a metaclass. Unfortunately, that may tremendously increase a model's complexity and, therefore, jeopardize its integrity and maintainability.

Multi-level modelling [1, 2, 11] aims at overcoming the limitations of the traditional object-oriented paradigm. It is characterized by the following key characteristics:

- Unlimited number of classification levels: A class can be defined on any classification level.
- Classes as objects: Every class is an object at the same time, that is, it may have a state and may execute operations.
- Deferred instantiation: Classes may define properties that apply not only to their direct instances, but to instances of instances of instances etc. Therefore, it is possible to specify that a property is to be instantiated only on a lower level.
- Classes on different classification levels may co-exist in one model: In the traditional paradigm, there is a strict distinction between model and modelling language. A multi-level model may include objects on different classification levels, which may located on M0 or M1 or may be part of languages or meta-languages.

3.2 Prospects of Using MLM for Modelling Product Lines

Modelling product lines is aimed at a clear definition of all relevant kinds of variability. To foster modelling productivity, a language for modelling product lines should promote reuse, which recommends powerful abstractions. Furthermore, such a language should support the maintenance of product lines, which recommends powerful abstractions, too. In ideal case, the concepts provided with the language should be suited for the entire software lifecycle, at least on different levels of precision and detail.

Classification is a powerful concept to express variability. A class defines the properties that are shared by all its instances. The property types define the extension of the set of instances. From another point of view a class can be regarded as a set of constraints that need to be satisfied by its instances. The more restrictive the properties, or more general: the constraints are, the better is the chance to clearly discriminate valid against invalid variants. Nevertheless, the variation that can be defined through classification is limited to the variation of instances of the class. Possible

variations of the class (and of other similar classes) cannot be accounted for. This is different with MLM. Metaclasses can be used to define variations of classes. In addition, they may serve to account for possible, future variability that is not known yet. For example, we know that types of bicycle frames are made of aluminum, carbon, and so on. Therefore, these materials can be used in a metaclass to define a range of possible variants. At the same time, we know that there may be other materials in the future, which can be accounted for by extending the set of materials at a later time. Hence, extending metaclasses on any level improves the chance to modify a product line in an elegant and convenient way.

3.3 Illustration: A Multi-Level Model of a Product Line

The diagram in Fig. 2 illustrates how variability can be represented in a multi-level model. In the example, we use the FMMLx [12], which is based on Xcore and implemented in the Xmodeler [98, 10]. Its concrete syntax indicates the level of a class by the background color of the class name field. The name of the metaclass is placed on top of the class name. Deferred instantiation is expressed by intrinsic features, where a feature may be an attribute, an operation, or an associated class. The instantiation level is defined through a white number printed on a black rectangle next to the name of a feature. In the case of intrinsic associations both sides of an association may have an instantiation level, which can be different. The state of an object (which may be a class at the same time) can be represented in a separate compartment (printed in green) as well as the values returned by operations (yellow on black).

The upper levels of Fig. 2 can be considered as models of products and corresponding software systems at different levels of abstraction. Software vendors will benefit from introducing (domain-specific) concepts in various levels and receiving those concepts all the way down the hierarchy of products. Moreover, the transition among levels is smooth: each level reduces the number of valid configurations of its upper level by introducing constraints or assigning values to attributes. The transition from M4 to M3, for example, is done by specifying that a racing bike, as well as its frame, is not suited for tough terrains but suited for races, a racing bike is suited for cities, a racing fork does not have a suspension, and so on. Note that the constraint marked as c1 in the figure could be specified in feature modeling, yet its specification would require complicated dependencies between features in the single level model.

The models in the MLM approach can be easily mapped into code as they already specify the type of the various elements. For example, a wheel, a frame, and a fork are classes, while size and weight are attributes. The three classes of bicycles, regular, racing, and pro-racers, are clearly shown in different abstraction levels of the MLM approach, M2, M3 and M4 in the figure, allowing their separation for different types of bicycle dealers, e.g., those who require differentiation between racing bikes and pro-racers (level M2) and those who do not require this (level M3).

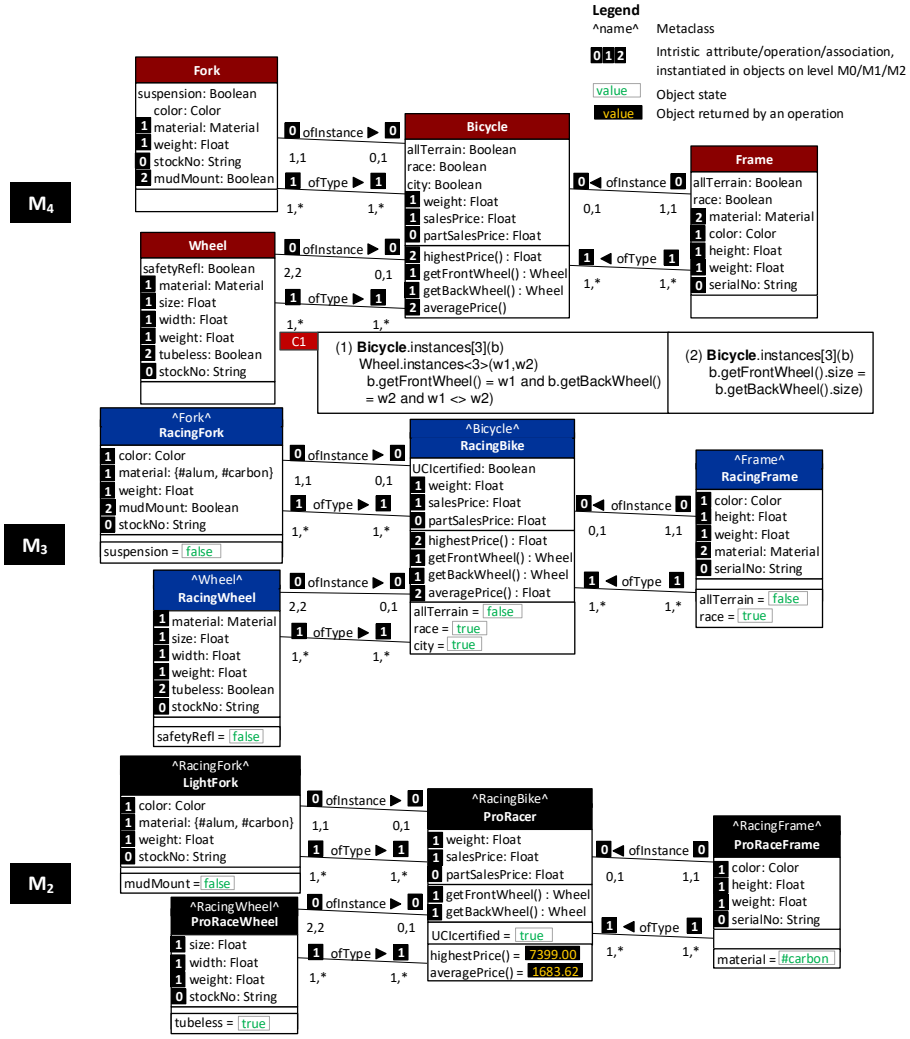

Fig. 2. A bicycle model in the MLM approach

4 Future Research

In this paper we suggest using MLM for specifying variability in software product lines in general and configuration in particular. We demonstrated MLM advantages over feature modeling which include better abstraction and expressiveness, unambiguity, and integrity. Those properties ease the reuse and maintenance of models and allows for easy transformation into later stages of design and implementation.

To take full advantage of the potential of the MLM approach, we plan to devise a multi-level constraint language which allows defining constraints that span over mul-

tiple levels of classification. In addition, we plan to develop a comprehensive method to guide the construction of multilevel DSMLs. While a method for developing DSMLs within the MOF paradigm provides useful support, it does not help with decisions concerning the appropriate number of levels or the separation of levels. We further intend to evaluate the usefulness of the approach for performing different SPLE activities, including constraining valid software configurations and deriving specific configurations to certain requirements.

References

1. Atkinson, C. and Kühne, T. (2008). Reducing accidental complexity in domain models. Software & Systems Modeling 7 (3), pp. 345-359.
2. Atkinson, C., Gutheil, M., and Kennel, B. (2009). A Flexible Infrastructure for Multilevel Language Engineering. IEEE Transactions on Software Engineering 35 (6), pp. 742–755.
3. Bashroush, R., Garba, M., Rabiser, R., Groher, I., and Botterweck, G. (2017). CASE Tool Support for Variability Management in Software Product Lines. ACM Comput. Surv. 50, 1, Article 14.
4. Bass, L., Clements, P., & Kazman, R. (2012). Software Architecture in Practice. SEI Series in Software Engineering, 3rd Edition.
5. Berger, T., Rublack, R., Nair, D., Atlee, J. M., Becker, M., Czarnecki, K., & Wąsowski, A. (2013, January). A survey of variability modeling in industrial practice. In Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (p. 7). ACM.
6. Chen L, Babar M A (2011) A systematic review of evaluation of variability management approaches in software product lines. Information and Software Technology 53: 344-362.
7. Czarnecki, K., Helsen, S., and Eisenecker, U. (2004). Staged Configuration Using Feature Models, Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004.
8. Clark, T. and Willans, J. (2012). Software language engineering with XMF and Xmodeler. Formal and Practical Aspects of Domain Specific Languages: Recent Developments. IGI Global, USA.
9. Clark, T., Sammut, P., Willans, J. (2008). Applied Metamodelling: A Foundation for Language Driven Development. Ceteva, 2 edition.
10. Clements, P. & Northrop, L. (2001). Software Product Lines: Practices and Patterns. Addison-Wesley.
11. De Lara, J., Guerra, E., and Cuadrado, J. S. (2014). When and How to Use Multi-Level Modelling. ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (2), Article no. 12.
12. Frank, U. (2014). Multilevel Modeling: Toward a New Paradigm of Conceptual Modeling and Information Systems Design. Business and Information Systems Engineering 6 (6), pp. 319–337.
13. Frank, U. (2014). Power-Modelling: Toward a more Versatile Approach to Creating and Using Conceptual Models. In Proceedings of the Fourth International Symposium on Business Modelling and Software Design.
14. Jacobson, I., Griss, M., & Jonsson, P. (1997). Software reuse: architecture, process and organization for business success. ACM, Addison-Wesley.

15. Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study (No. CMU/SEI-90-TR-21). Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.
16. Pohl, K., Böckle, G., van der Linden, F. (2005) Software Product-line Engineering: Foundations, Principles, and Techniques, Springer.
17. Prehofer, C. (1997). Feature-oriented programming: A fresh look at objects. ECOOP'97—Object-Oriented Programming, 419-443.
18. Reinhartz-Berger, I., Sturm, A., & Clark, T. (2015). Exploring Multi-Level Modeling Relations Using Variability Mechanisms. In MULTI@ MoDELS, pp. 23-32.
19. Van Gurp, J., Bosch, J. & Svahnberg, M. (2001). On the notion of variability in software product lines. In proceedings of the Working IEEE/IFIP Conference on Software Architecture, pp. 45-54.