Carrying Ideas from Knowledge-based Configuration to Software Product Lines

Juha Tiihonen[1], Mikko Raatikainen[2], Varvana Myllärniemi[2], and Tomi Männistö[1]

[1] {firstname.lastname}@cs.helsinki.fi, University of Helsinki, Finland
[2] {firstname.lastname}@aalto.fi, Aalto University, Finland

Abstract. Software variability modelling (SVM) has become a central concern in software product lines – especially configurable software product lines (CSPL) require rigorous SVM. Dynamic SPLs, service oriented SPLs, and autonomous or pervasive systems are examples where CSPLs are applied. Knowledge-based configuration (KBC) is an established way to address variability modelling aiming for the automatic product configuration of physical products. Our aim was to study what major ideas from KBC can be applied to SVM, particularly in the context of CSPLs. Our main contribution is the identification of major ideas from KBC that could be applied to SVM. First, we call for the separation of types and instances. Second, conceptual clarity of modelling concepts, e.g., having both taxonomical and compositional relations would be useful. Third, we argue for the importance of a conceptual basis that provides a foundation for multiple representations, e.g., graphical and textual. Applying the insights and experiences embedded in these ideas may help in the development of modelling support for software product lines, particularly in terms of conceptual clarity and as a basis for tool support with a high level of automation.

Keywords: Variability modelling, Feature modelling, Knowledge-based configuration, Conceptualization, Variability management

1 Introduction

Software product lines (*SPL*) have emerged as an important means for reuse in the context of a set of products that share a common *SPL architecture* and other assets (e.g. [5]). For SPLs, variability management has become a central concern. *Variability* is the ability of a system to be efficiently extended, changed, customised or configured for use [19]. *Domain engineering* develops assets for reuse while exploiting reusable commonalities and catering for differentiating variability. *Application engineering* realises the products of a SPL by reusing the assets, by resolving the variability, and by developing product specific extensions. *Software variability modelling* (*SVM*) represents the variability of the assets.

A special class of SPLs is a *configurable software product line* (CSPL), in which all differences between the product variants have been pre-defined and

Author's Post-print: Submitted author final version post-refereeing.
Tiihonen, J., Raatikainen, M., Myllärniemi, V., & Männistö, T. (2016). In M. G. Kapitsaki & E. de Almeida (Eds.), Software Reuse: Bridging with Social-Awareness: Proceedings of the 15th International Conference on Software Reuse, ICSR 2016, pp. 55–62. Limassol, Cyprus, June 5-7, 2016, Springer International Publishing, Cham, Switzerland. doi:10.1007/978-3-319-35122-3_4

implemented in domain engineering. Product derivation involves merely making decisions on the predefined and implemented assets and variability therein [3]. This specification of an individual product is also called a *configuration* for short (cf., [7]). Recently, CSPLs have received increasing attention in different forms: Dynamic SPLs, the application of SPL to autonomous or pervasive systems, and service oriented SPLs are examples where the idea of CSPL can be applied.

In the field of physical, such as mechanical products, *knowledge-based configuration* (*KBC*) (e.g., [10]) is a related domain to SPL in general, and CSPL in particular. KBC aims to model and manage variability in a way that enables automated product derivation. Besides similarity, the long history since 1980's and relative maturity makes KBC an interesting field to compare with CSPLs.

Compared to previous work [14, 1, 12], we aim to investigate synergies between KBC and SVM in more depth and from the variability modelling point of view. The research problem of this paper is: *What ideas from knowledge-based configuration can be applied to software variability modelling and configuration?* We highlight three ideas of KBC and discuss their potential implications for SVM in general and especially in the context of CSPLs.

In terms of the methodology, our analysis and comparison of the literature focuses on core KBC modelling literature and feature models that are the most common modelling method of SVM. A search based on title and abstract through all special issues on configuration and the proceedings of configuration workshops since 2000 was performed to augment already known relevant KBC literature on modelling conceptualisations. We focus on aspects of variability modelling that are relevant to supporting product derivation and configurability.

The rest of this paper is organised as follows. Section 2 identifies previous work. Section 3 presents the three identified potentially useful ideas from KBC. Section 4 provides discussion and concludes.

2 Previous work

Knowledge Based Configuration emerged from various domains of physical products such as computers and elevators. It is a relatively general, widely deployed and domain-independent approach with quite a long history [10]. The core of knowledge representation in KBC forms two widely cited and fundamentally similar conceptualisations of Soininen et al. [18] and Felfernig et al. [9]. In the conceptualisation of Soininen et al. *configuration model knowledge* specifies the entities that can appear in a *configuration* specifying an individual product, their properties, and the rules on how the entities and their properties can be combined. *Individuals* (instances) of configuration model concepts describe individual configurations and thus represent configuration solution knowledge. Finally, *requirements knowledge* specifies the systematised requirements on the configuration to be constructed. Besides being widely cited by researchers, these types of configuration knowledge representations are "typically provided in today's commercial configuration environments" [11].

Software Variability Modelling has been elevated as a central concern for SPLs in addition to reuse [4]. Feature modelling (*FM*) is probably the first and the most widely known means to represent SPL variability. A *feature* represents a characteristic of a system that is visible to the end-user [13], or in general, a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among product variants [6]. Other variability modelling approaches include the orthogonal variability modelling approaches such as OVM [15] that define a separate model that is associated with the base model such as an UML model; and decision-oriented [16] approaches model variability as questions and possible answers to be presented in the style of wizards.

3 Ideas from Knowledge Based Configuration

3.1 I1: Separating Types and Instances

In KBC, domain and application models are clearly separated: domain models are expressed as types (that are instances of a meta-model) and application models as instances of the types.

SPL engineering makes a clear distinction between domain and application engineering activities [5]. However, most of the research on variability modelling seems to focus on domain engineering and variability representation; and application engineering has often remained more implicit [7].

Conceptual separation of domain and application models. In feature modelling, there is no clear difference between a domain model and an application model, but the same modelling concepts are used for both purposes. This is illustrated in Fig. 1. Instantiating a product feature model takes place by *specializing* the product line feature model. Each operation in resolving variability results in another feature model containing less variability. When all variability has been resolved, the remaining features represent the valid, fully resolved (*specific*) configuration [6]; for an example see the lower part of Fig. 1. The only way to recognize that a feature model describes a product variant is to investigate whether all variability has been resolved. Additionally, it can become challenging to differentiate the product line feature model from a series of specialized models or to distinguish evolution of the variability models from their specialization.

In KBC, a clear separation between domain and application models is made. Fig. 2 illustrates how a product line feature model and a product feature model could be represented. In the product line feature model, modelling concepts are called *feature types*, whereas modelling concepts in the product feature model are called *feature instances*. Instead of specializing, the model of the product is *instantiated* from the model of the product line. In KBC terminology, a generic description of a product family (*configuration model*) is instantiated into an unambiguous specification of a concrete product individual (*configuration*). Consequently, feature types in the configuration model are instantiated as feature instances in the configuration (Fig. 3). When following the distinction between types and instances, the different levels of feature modelling can be seen as instantiations: modelling concepts are instantiated as concrete feature types in the

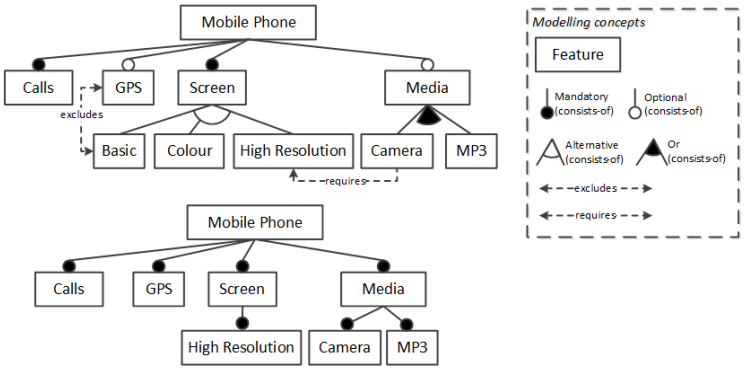

Fig. 1. A sample feature model [2]. Same modelling concepts are used to represent both the product line feature model (top) and product feature model (bottom); the latter is specialized from the former.

domain models, which are then instantiated as concrete feature instances in the application models (Fig. 3).

Types modularize models and facilitate reuse. Besides conceptual differentiation, there are other advantages to apply types and instances. A type declaration provides a convenient means for modularizing a reusable asset so that each logical entity can be defined and managed independently. A type is a natural place to collect specifications of compositional structure, attributes, constraints, and other modelling constructs. The set of type declarations forms a repository of reusable assets. The types can then be reused in the context of larger entities and eventually to model an entire SPL. Another advantage of types and instances is the reuse of a type within a product of a product line via instantiation. This seems conceptually cleaner than *referencing* and *cloning* suggested for feature models [6].

3.2 I2: Conceptual Clarity

In KBC, there are two main relations: classification (*is-a*) and composition (*has-part*) with *cardinality* to expresses compositional rules such as mandatory or optional. With these, two respective major hierarchies emerge. Composition can pick e.g. alternatives from various branches of the classification hierarchy.

Initially, FODA [13] defined *features* and *mandatory*, *optional* and *alternative* relations between features, along with *mutually exclusive with* and *requires* constraints (Fig. 1). Over time, the need for representing more complex variability has emerged. For example, there is a need to represent *or* (Fig. 1), which indicate that one or more features from the child features must be included [2]. The exact nature of the modelling concepts should be explicit and unambiguous.

Distinct relationships such as *has-part* and *is-a*. In the context of FM, particularly, the *alternative* relation has proved to be difficult to interpret

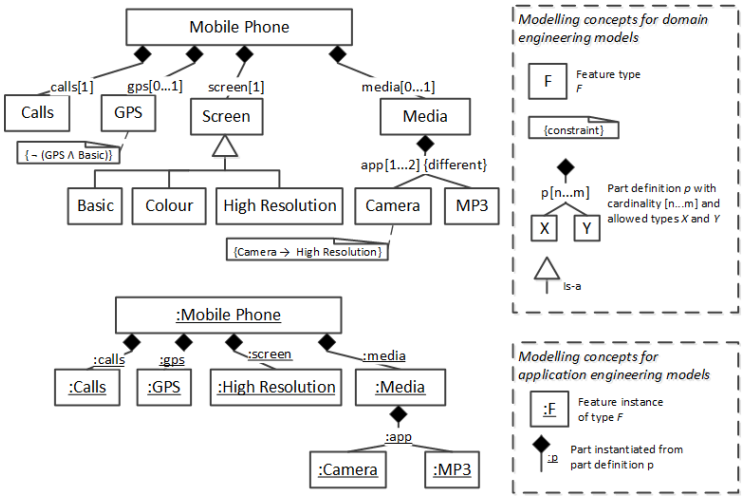

Fig. 2. The sample feature model in Fig. 1 represented to follow the KBC conceptualizations: a distinction between domain and application models is made. The features in the product model are instantiated from the feature types in the product line model.

(Fig. 4, a,b). Originally, this relation denoted *specialisation*: "[a]lternative features can be thought of as specializations of a more general category" [13]. This is concretely manifested by the *alternative* relation in Fig. 4 (a): the domain engineer does not read the model as "*the mobile phone consists of one screen, and the screen consists of basic, colour or high resolution screen*". Instead, the obvious intention is that "*the mobile phone consists of one screen, and the screen can be a basic, colour or high resolution screen.*" (Fig. 4, b).

To organise and specialise types, KBC adopts classification (*is-a*) and inheritance of features in the usual object-oriented manner. For example, *Screen* can be specialised into *Basic*, *Colour* and *High Resolution* screens (Fig. 2).

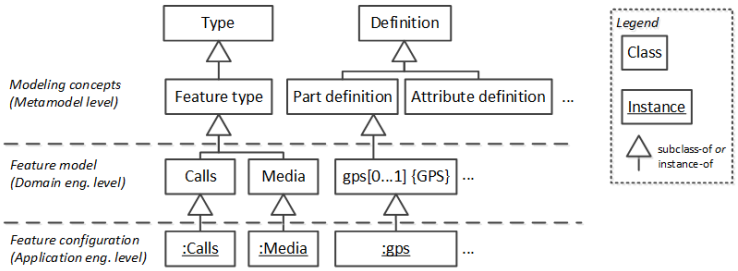

Fig. 3. Following the KBC conceptualizations, feature modelling involves three instantiation levels: modelling concepts, product line feature models with feature types, and product feature models with feature instances. Adapted from [18].

Cardinality as a basis for compositional relationships. In the context of FM, composition is the fundamental relationship and the need to represent different kinds of cardinalities [6] has been identified. However, instead of replacing *mandatory*, *optional*, *alternative* and *or* with cardinalities, cardinalities are feature model extensions Fig. 4 (c,d). That is, instead of *refining* previous modelling conceptualisations, the existing conceptualisations are *extended* by adding new concepts on top of the old ones.

In KBC, a means to model varying compositional structure is via *part definitions* [18] that include cardinalities. In the example of Fig. 2, type *Mobile phone* has a part definition *calls[1]* of allowed type *Calls*: one feature *Calls* must be present in a valid configuration. As another example, type *Media* has a part definition *apps[1...2]* of allowed types *MP3* and *Camera*. The semantics of a part definition is that in a configuration, a valid instance of the whole type has the number of part instances specified by the *cardinality* as parts with the specified *part name*; each instance as a part must be of one of the *allowed types*. Note that allowed types do not have to be subtypes of the same type. Naturally, it is possible to reuse a type as an allowed type in several contexts.

3.3 I3: Separate Domain Phenomena, Concepts and Representations

The modelling concepts in KBC are defined and provided with semantics independently of the representations of concepts.

SVM in terms of FM started with graphical feature diagrams (cf., [17]). Numerous dialects of graphical FM notations have been proposed [2, 17] and even textual FM languages have emerged (cf., [8]) — some of these also introduce new concepts. The full semantics of the concepts or notations has also been provided, although often as an afterthought [17]. However, two concerns are combined: what are usable or otherwise appropriate representations and what phenomena a model needs to capture.

Domain *phenomena* as *concepts* with semantics. Well-defined concepts are the fundamental basis for capturing the phenomena of the domain. They are an asset on which representation formats for various (but similar) purposes can be developed. In KBC, modelling conceptualisations have been defined

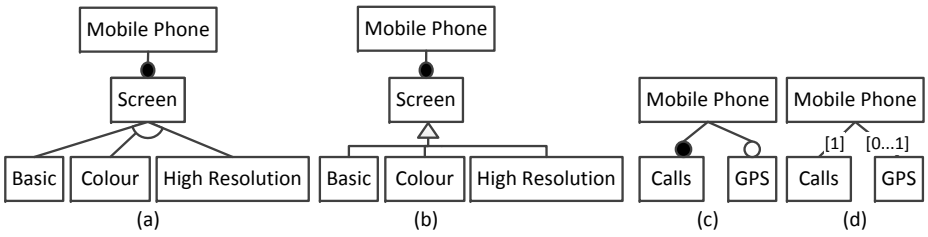

Fig. 4. (a) Alternative originally implied specialisation. (b) Idea to model specialisation as is-a, not consists-of. (c,d) Extending but not refining FM concepts: consists-of with cardinality could have replaced mandatory, optional and other consists-of relations.

independent of direct representations [18, 9]. The idea is that concepts can be defined independently of representations so that appropriate concepts capture the domain phenomena aptly.

Multiple equivalent representations of concepts. When concepts capture domain phenomena, it is more straightforward to support multiple representations than when attempting to directly capture the domain phenomena or to perform model transformations between representations. First, it is easier to have simultaneous representations when the representations are based on the same concepts. Second, changes or adaptations such as shorthand notations or semantic sugar are easier to add to representations without affecting other representations. Consequently, completely new representations are easier to add. An example from KBC is described in [20] where a model is kept in internal data structures and it can be edited in textual representation, as called for also in SVM (cf. [8]), and via a graphical editor.

4 Discussion and Conclusions

We explored the research in Knowledge-Based Configuration (KBC) to identify major ideas that could be applied to Software Variability Modelling (SVM), particularly in Configurable Software Product Lines. We reflected the ideas to the existing research in SVM with the hope that the ideas could provide fresh insight and novel ideas for advancing the state of the art and practice in SVM. This analysis was performed assuming that automation is desired – the ideas might not fit less rigorous SVM, e.g., when exploring the variability of a domain. Fully exploiting some of the benefits requires tool support, e.g., to benefit from multiple representations of the concepts or modelling with types and instances.

We argue for having separate models for domain and application engineering, i.e., separate models for the product line and for product variants. Further, we see room for re-factoring the feature modelling concepts and relationships for better conceptual clarity. This could apply types and instances as a mechanism to simplify the reuse of assets within and between the products of a product line. The separation of well-defined concepts from representations makes the management of variability models more straightforward. If a model has a well-defined conceptualisation with declarative semantics, a straightforward translation can be carried out to produce an equivalent model that can be reasoned upon.

Some of the ideas are already reflected in some SVM approaches, but not in mainstream SVM. However, the ideas address interrelated concerns – full benefits stem from being applied simultaneously.

Future work can identify additional ideas that can be applied to SVM from KBC and vice versa. Concretising, refining and extending the ideas into conceptualisations, representations and supporting tools would enable practical utilisation. Both theoretical and empirical research is needed, e.g., conceptual refactoring would benefit from empirical investigation on what concepts are needed to form a clear conceptual foundation that is neither too minimal nor bloated.

References

1. D. Benavides, A. Felfernig, J. Galindo, and F. Reinfrank. Automated Analysis in Feature Modelling and Product Configuration. In J. Favaro and M. Morisio, editors, *ICSR*, number 7925 in LNCS, pages 160–175, Pisa, Italy, 2013. Springer.
2. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.
3. J. Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In G. J. Chastek, editor, *SPLC '02*, pages 257–271, 2002.
4. L. Chen and M. Ali Babar. A systematic review of evaluation of variability management approaches in software product lines. *IST*, 53(4):344–362, 2011.
5. P. Clements and L. Northrop. *Software product lines practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
6. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process Improvement and Practice*, 10(1):7–29, 2005.
7. S. Deelstra, M. Sinnema, and J. Bosch. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):173–194, 2005.
8. H. Eichelberger and K. Schmid. Mapping the design-space of textual variability modeling languages: a refined analysis. *International Journal on Software Tools for Technology Transfer*, Dec. 2014.
9. A. Felfernig, G. E. Friedrich, and D. Jannach. UML as Domain Specific Language for the Construction of Knowledge-based Configuration Systems. *International Journal of Software Engineering and Knowledge Engineering*, 10(4):449–469, 2000.
10. A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen, editors. *Knowledge-Based Configuration: From Research to Business Cases*. Morgan Kaufmann, 2014.
11. L. Hotz, A. Felfernig, M. Stumptner, A. Ryabokon, C. Bagley, and K. Wolter. Configuration Knowledge Representation and Reasoning. In A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen, editors, *Knowledge-Based Configuration: From Research to Business Cases*, pages 41–72. Morgan Kaufmann, 2014.
12. A. Hubaux, D. Jannach, C. Drescher, L. Murta, T. Männistö, K. Czarnecki, P. Heymans, T. Nguyen, and M. Zanker. Unifying software and product configuration: A research roadmap. In *ECAI 2012 Workshop on configuration*, pages 31–35, 2012.
13. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis Feasibility Study (FODA). Technical Report CMU/SEI-90-TR-021, Carnegie Mellon U., Software Engineering Institute, Pittsburgh, PA, USA, 1990.
14. T. Männistö, T. Soininen, and R. Sulonen. Product configuration view to software product families. In *ICSE Software Configuration Management Workshop*, 2001.
15. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
16. K. Schmid, R. Rabiser, and P. Grünbacher. A comparison of decision modeling approaches in product lines. In *5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 119–126, New York, NY, USA, 2011. ACM.
17. P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Compututer Networks*, 51(2):456–479, 2007.
18. T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a general ontology of configuration. *AI EDAM*, 12(04):357–372, 1998.
19. M. Svahnberg, J. van Gurp, and J. Bosch. A taxononomy of variability realization techniques. *Software—Practice and Experience*, 35(8):705–754, 2005.
20. J. Tiihonen, M. Heiskala, A. Anderson, and T. Soininen. WeCoTin–A practical logic-based sales configurator. *AI Communications*, 26(1):99–131, 2013.