MASS CUSTOMIZATION FOR EVOLVING PRODUCT FAMILIES

Thorsten Krebs, Katharina Wolter

Laboratory for Artificial Intelligence
University of Hamburg
Vogt-Köln-Str. 30
Germany, Hamburg, 22527
Thorsten Krebs, krebs@informatik.uni-hamburg.de
Katharina Wolter, kwolter@informatik.uni-hamburg.de

Lothar Hotz

HITeC c/o
University of Hamburg
Vogt-Köln-Str. 30
Germany, Hamburg, 22527
Lothar Hotz, hotz@informatik.uni-hamburg.de

Abstract: Evolution of products is inevitable throughout their life cycle – driven by advancing technology, increasing customer requirements or bug fixes. Therefore, the set of components as well as the dependencies between those components are getting more complex and only few experts are able to configure products. But in mass customization scenarios it is desired to generate products specific to customer requirements. One promising approach to this is abstracting from the available components and their interdependencies by focusing on features. In an extended configuration process, the customer can select a set of features for the desired product and based on pre-defined mappings the system architecture is selected and the corresponding product components are inferred. This approach hides modifications to product components by showing the same or only slightly changed features to the customer.

Significance: The work presented in this paper shows a basic method that can be adopted for tool support concerning evolution of configuration models. Further research in this area and automated support for evolution of configuration models is expected.

Keywords: Mass Customization, Product Families, Features, Configuration Models, Evolution

(*Received* ; *Accepted*)

1. INTRODUCTION

Product configurators are widely used to generate customer-specific products out of a vast amount of potential variants. This combines the two essentials of mass customization: adapting the derivation process by producing standardized assets and assembling these assets to make the unique product for the customer.

Evolution of products and product components is inevitable throughout their life cycle – driven by advancing technology, increasing customer requirements or bug fixes. Therefore, the set of components as well as the dependencies between those components are getting more complex and only few experts are able to configure products based on such complex configuration models. But in mass customization scenarios it is desired to generate products specific to customer requirements. Thus, the configuration process would largely benefit when everybody is able to configure products – e.g. in internet configuration scenarios (compare [Ardissono et al. 2002]).

One promising approach to this is abstracting from technical details of the available components and their interdependencies by focusing on functionality. Functionality of products and product components can be represented with features (see also [Kang et al. 1990]). In general, the expected improvements of using a feature-based approach for software development are better control over variability, extended reuse of requirements, improved configuration support and sales support as well as improvements on new development [Hein et al. 2000].

The structure-based configuration method [Günter 1995] can be used to support the feature-based approach. In such a feature-based configuration process the customer can select a set of features for the desired product and based on pre-defined mappings between features and the product line artifacts a (potentially automated) process constructs the system

architecture by selecting the corresponding product components (hidden for the user). This approach gives the possibility of hiding modifications to product components by showing the same or only slightly changed features to the customer.

The remainder of this paper is structured as follows: in Sections 2 we show how features and their interrelations can be described in detail. Next, we explain how features, software artifacts, hardware artifacts and the mapping between them can be defined in the configuration model (Section 3). How this configuration model is used to derive complex products is explained in Section 4. Finally, the aspect of evolving product components is introduced and change operations used to capture the modification of configuration models are described (Section 5). A short outlook concludes the paper.

2. FEATURES

We follow the definition in [Kang et al. 1990] where a feature is defined as "a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems". Further, "a feature model is a specific domain model covering the features and their relationships within a domain" [Ferber et al. 2002]. Features are modeled hierarchically in an ontology that can consist of taxonomy and partonomy. Taxonomic relations (i.e. specializations) allow defining more specific system properties and therefore a distinction between different types of features. Compositional relations provide the means for grouping related features by placing them next to each other (subfeature relation). Feature hierarchies typically contain facilities to express diverse types of variability:

- *Mandatory* features are present in all products that belong to the product family in the modeled domain.
- *Optional* features may or may not be included in a product. If an optional feature is not part of the product, all subfeatures of that feature are also excluded.
- *Alternative* features represent a choice between a set of features from which exactly one has to be chosen.
- *Multiple* features capture the possibility to choose multiple features from a set of features, but at least one has to be chosen.

Properties of product lines are common to all product line members – i.e. they are mandatory features. Mandatory features represent basic functionality of all products in the given domain. Optional features are not included in all but only in some selected products and represent the admissible differences between product line members.

Because features can be interconnected not only by hierarchical relations (i.e. taxonomies and partonomies) but also by dependencies concerning arbitrary features, more sophisticated variability mechanisms are needed. The following is a list of further needed dependencies (compare [Brüne et al. 2003], [Krebs and Hotz 2003]):

- *Requires:* Features can be required by other features - i.e. the existence of the required feature is needed for the former one.
- *Excludes:* Features may exclude each other. This happens when two features can not be selected together, e.g. when the system components that realize these features are incompatible. This is a mutual exclusion.
- *Recommends:* A weak form of the requires relation is a recommendation. The existence of features can be recommended for other features. This can also be seen as the semantics of a default value.
- *Discourages:* Contrary to the latter, features may be discouraged for other features in the system. This is a weak form of the mutual exclusion. Hence, it describes that a feature is not chosen per default.

Due to views of various granularities, features are utilized to model different levels of abstraction. [Kang et al. 2002] e.g. distinguish between product *capabilities*, the *operating environment*, *domain technologies* and *implementation techniques*. While product capabilities are general terms that also customers can understand and select the desired functionality from, implementation techniques are usually hidden from customers and used by application engineers that implement products or product artifacts. Therefore, mappings between features on the diverse levels can be modeled through taxonomic and compositional relations and dependencies.

3. MODELING WITH STRUCTURE-BASED CONFIGURATION

In product families besides features also artifacts (i.e. software modules and hardware components) are considered. In our approach features, artifacts and the mapping between these are formalized in a *configuration model* based on mechanisms from structure-based configuration. This model is further used for automatic product configuration. Features, software modules and hardware components can be represented with *concepts* provided by the modeling facilities of structure-based configuration. A definition is given in the following:

- Each concept has a *name* which represents a uniquely identifiable character string.
- Concepts are related to exactly one other concept in the taxonomic relation. Thus, the latter concept is the *superconcept* of the former one.
- Attributes of concepts can be represented through *parameters*. A parameter is a tuple consisting of a name and a value descriptor. Diverse types of domains are predefined for value descriptors – e.g. integers, floats, sets and ranges.

- Partonomies are generated by modeling *compositional (has-parts) relations*. Such a relation definition contains a list of parts (i.e. other concept definitions) that are identified by their names. Each of these parts is assigned with a minimum and a maximum cardinality that together specify how many instances of these concepts can be instantiated as parts of the aggregate.
  Has-parts relations are a class of compositional relations. Thus, the knowledge engineer is free to define his own relations and give his own names – e.g. a concept can contain a has-features definition next to a has-parts definition to emphasize on the difference between artifacts and features.

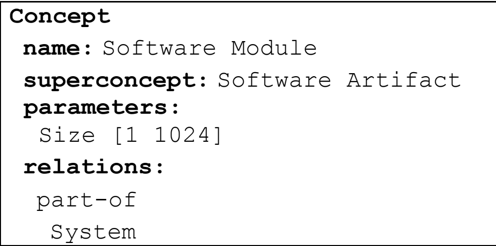

Figure 1. Concept Definition

Concepts can be utilized to model all kinds of entities of the product family like software and hardware artifacts or features. Through the taxonomic relation these entities are grouped next to each other in one configuration model – each having their own branch in the hierarchy. Therefore, we have defined a *common applicable model (CAM)* which contains predefined concepts with properties common to all these entities. An earlier version – called *upper model* – was already introduced in [Hotz and Krebs 2003]. In Figure 1 the definition of the concept *software module* is depicted. The entry *superconcept: software artifact* indicates that the software module is taxonomically placed under the concept software artifact which is predefined in the common applicable model. One parameter is defined: the *size* of the software that can range between one kilobyte and one megabyte. Further an inverse definition of the compositional relation (i.e. *part-of*) is given to express that this *software module* belongs to a *system*.

Figure 2 shows the CAM that contains definitions for the basic entities *feature*, *hardware artifact*, *software artifact* and the *system* that represents configurable product family members and contains the configurable artifacts through compositional relations. Application-specific concepts are introduced into the configuration model by simply placing them under the given entities – e.g. a *software module* is introduced by taxonomically relating it to the concept definition of the *software artifact* (*software artifact* is superconcept of the new *software module*).

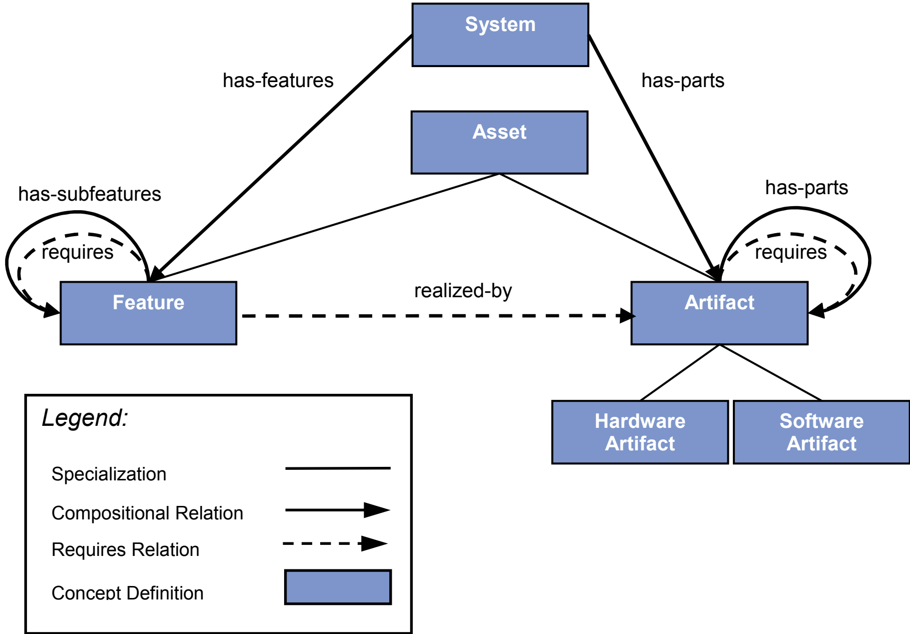

Figure 2. Common Applicable Model (CAM)

Further relations between assets that do not describe a hierarchical structure like taxonomic and compositional relations do are called *dependency relation*. A *uni- or bidirectional* dependency can be defined between concepts in arbitrary places of the ontology. Examples for this relation type are the *requires* relation for unidirectional und the *excludes* relation for bidirectional dependencies. For bidirectional relations this means that during the configuration process this relation is processed in both ways, no matter which concept participating in the relation is instantiated first. Unidirectional relations are different; they are only processed in one direction. E.g. when a software module requires the existence of a specific driver the existence or absence of this driver has no impact on that software module.

The *realizes* relation is a further bidirectional dependency that expresses that features are realized by artifacts in an *n*-to-*m* mapping: one feature can be realized by one or more artifacts and the other way round one or more features can be realized by one artifact. Several situations during product derivation can occur when handling *n*-to-*m* mappings. The easiest possibility is when a feature is given and the needed artifacts descriptions are generated. A more complex situation is given when an artifact is given (e.g. computed by a feature) that realizes a further feature. This feature has to be generated and the artifact has to be integrated. A even more complex situation is given, when a feature is realized by several artifacts like in the first situation, but one or more of these artifacts are already related to the feature, others are already generated but not yet integrated (because they are used by other features) and further artifacts are not yet generated. For those tasks a clear separation of possible features and artifacts on the one side and features and artifacts that are used for a specific product derivation on the other side is needed. In the structure-based configuration tools like KONWERK [Günter and Hotz 1999] this is realized by distinguishing between *concepts* for describing features and artifacts in general and *concept instances* for describing features and artifacts of a specific product derivation. In situations as they are described above for some features concept instances and several artifacts are already generated and for not yet generated artifacts still concept description are present.

4. PROCESS

The configuration process in structure-based configuration is an incremental process – in each step one configuration decision is made and its effects on the partial solution configured so far are computed. Thus, in an interactive configuration process some decisions are made by the user and some are inferred by the configuration system. One aim of our approach is that most of the decisions on a lower level of abstraction (e.g. decisions about components or their parameters) are inferred by the configuration system. This is possible because of the mapping between features and artifacts defined in the configuration model (see Figure 2).

The first step in the configuration process is the selection of the product one wants to configure. Next, its features are selected. For each feature the user selects the configuration system infers the hardware and software artifacts necessary to realize the feature based on the realized-by relation in the configuration model. Thus, the product configuration is more efficient because the user only has to make those decisions that cannot be inferred based on other decisions already made. Furthermore, feedback concerning the effects of decisions can be given after each decision made by the user. For example, the user can realize that a specific feature cannot be selected anymore because of earlier decisions. This is especially important for non-expert users who are not familiar with the dependencies between decisions in the configuration process. However, even for experts this is a useful support, since they cannot overlook dependencies by mistake during the configuration process.

Based on the configuration model it is possible to compute all necessary decisions needed to configure a specific product. These decisions are collected in an agenda and can be displayed to the user. Additionally, these decisions can be grouped in several *subtasks* to structure the process. One group e.g., can encompass all decisions concerning features and further groups consist of decisions about hardware artifacts and software artifacts. This structuring of the configuration process is also defined in the configuration model, in a specific part called procedural knowledge. Since a configuration process can consist of a large number of decisions it is important to guide the user during configuration. Displaying these subtasks (in a specific order) to the user is one possibility to provide guidance.

5. EVOLUTION OF THE MODEL

Products and / or product components evolve over time. New version and variants are built, and as a fact of synchronization between the configuration model and the existing artifacts, also modeled for automated product derivation. This can have an effect on the relations between the artifacts that are combined for generating specific products as well as the relations between features and artifacts. A constantly growing artifact repository increases cognitive complexity and makes it more difficult to configure products specific for a given task description.

For most customers, however, it is not of interest which versions of software modules are used in the product. The functionality plays the key role in selecting the product that best suits the customer's requirements. Therefore, it is natural that evolution of artifacts (shown in terms of version numbers etc.) should be hidden from the customer. In most scenarios, especially in embedded systems like car electronics or mobile phones, the product that satisfies a required functionality is sufficient and the version numbers of software modules are not relevant. Such a behavior can be achieved by maintaining

the mapping between features and artifacts that realize those features during evolution of the model. When for example a new version of a software artifact is generated, the mapping(s) between this artifact and the feature(s) that this module realizes are simply modified such that they now are linked with the new version. After this, during an automated configuration process the new version will be selected. This evolution of the mapping between features and artifacts is depicted in Figure 3.

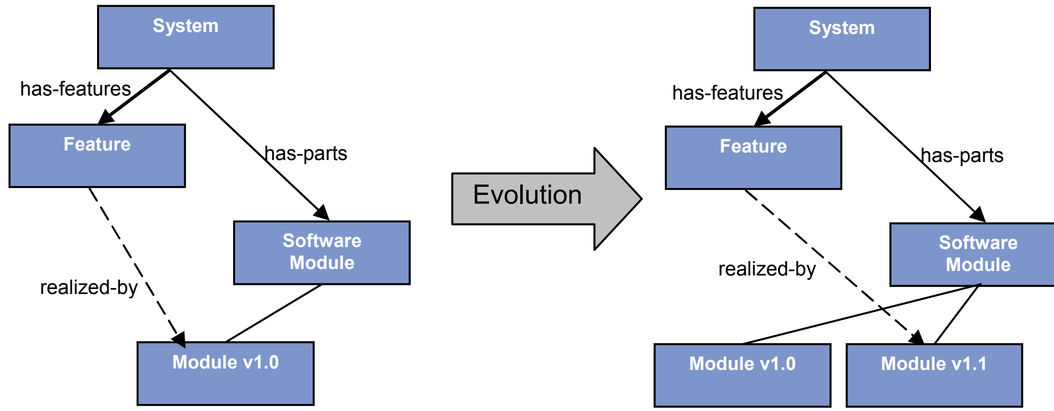

Figure 3. Synchronized Evolution of Artifacts and Mappings

Modifications to the configuration model are captured through *change operations*. *Basic* operations are those that cannot be further partitioned – i.e. simple modifications like adding a parameter value or deleting a concept definition. *Complex* change operations are composed of multiple basic operations or include some additional knowledge about the modification. They provide a mechanism for grouping basic operations into a logical unity (compare [Klein and Noy 2003]). Moving a group of concept definitions to a different superconcept for example can be split into moving each of those concepts separately. Hierarchies of change operations can be formulated to exploit the inheritance mechanism for specifying common properties.

Based on the assets and the relations that are specified in the CAM the complex change operations can be composed to force a combined modification of dependent modifications. Adding features or adding artifacts e.g. have direct impacts on the consistency of the mappings as further elaborated in the following.

- *Add feature:*
  Features describe the functionality of the product or of product components. Therefore, a feature that is not realized by some component is useless for product derivation[1]. Instead, adding a feature to the configuration model should directly entail the addition of the mapping between this feature and the artifact(s) that realize(s) it. The complex change operation can e.g. look like this:

  add feature -> (introduce feature, add mapping(s) to artifact(s))

- *Add artifact:*
  When new artifacts (in form of a concept definition) are introduced into the configuration model, they inherit the relations from their superconcept definitions. For dependencies (e.g. excludes or realizes), however, this is not always applicable. A new version of a software module may contain a bug fix and therefore no longer stay in conflict to other artifacts like the previous version did – modeled through an excludes dependency. The new version of that module usually realizes the same feature(s) the old version did, but it may also realize additional ones or be seen as a replacement for future products. For new versions (assuming that the newest version should be used) usually the mapping should be moved while for new variants (that are intended to coexist) a copy of the mapping has to be introduced. This results in the following complex change operations:

  add new artifact -> (introduce artifact, add mapping(s) to feature(s))
  add version -> (introduce artifact, move mapping(s) to feature(s))
  add variant -> (introduce artifact, copy mapping(s) to feature(s))

[1] We do not consider *planed features* in this paper.

The relations for the basic entities feature and artifact are predefined in the CAM and therefore can be automatically processed. Furthermore, some kind of dependency analysis is used to inspect other relations and restrictions that have been added to more specific concept definitions for building the complex change operation.

6. CONCLUSION & OUTLOOK

Using feature models has been widely implemented to improve product modeling for cognitive reasons [Felix et al. 2001] (documentation and customer-sales scenarios), product derivation [Hein et al. 2001; Kang et al. 2002] (supporting developers and enhancing reuse strategies) and configuring products in a more customer-oriented way [Ardissono et al. 2002; Felfernig et al. 2002] (e.g. in internet scenarios).

For feature-based configuration models, complex change operations can give the needed means for achieving that modifications to features and / or artifacts and mappings between features and artifacts are always processed together. Therefore, in a tool that supports evolution of configuration models, the modification of a feature or artifacts cannot be done independently of inspecting the relations this artifact participates in. Instead, complex change operations are composed that always have to be processed as one operation. Thus, the user can be forced to also modify the corresponding mappings or in a more sophisticated reasoning engine this might be (at least partially) automated.

The underlying dependency analysis is expected to be also usable for similar tasks like innovative configuration. This method can be used in conflict situations – i.e. in situations where the configuration solution generated so far, the configuration model and the task specification are not consistent. *Innovative configuration* is the task of extending the configuration model dynamically in the configuration process such that newly introduced concepts or concept properties enable additional solutions to the given task [Hotz and Vietze 1995]. A danger here is that dependencies to other concept definitions and configuration decisions can be violated.

7. ACKNOWLEDGMENTS

This research has been supported by the European Union under the grant IST-2001-34438, ConIPF - Configuration in Industrial Product Families.

8. REFERENCES

1. Ardissono, L., Felfernig, A., Friedrich, G., Goy, A., Jannach, D., Meyer, M., Schäfer, R., Schütz, W. and Zanker, M. (2002). Customizing the Interaction with the User in Online Configuration Systems. Proceedings of the Configuration Workshop on 15th European Conference on Artificial Intelligence (ECAI 2002), Lyon, France, pp. 119-124.

2. Brüne, S., Halmans, G. and Puhl, K. (2003). Modelling Dependencies between Variation Points in Use Case Diagrams. Proceedings of the 9th International Workshop on Requirements Engineering Foundation for Software Quality (REFSQ), Klagenfurt, Austria.

3. Felfernig, A., Friedrich, G., Jannach, D. and Zanker, M. (2002). Semantic Configuration Web Services in the CAWICOMS Project. Proceedings of the Configuration Workshop (ECAI 2002 Workshop), pp. 82-88, Lyon, France.

4. Felix, D., C. Niederberger, P. Steiger, and M. Stolze (2001). Feature-oriented vs. Needs-oriented Product Access for Non-Expert Online Shoppers. Proceedings of the 1st IFIP Conference on e-commerce, e-business, and e-government (I3E), pp 399-406.

5. Ferber, A., Haag, J. and Savolainen, J. (2002). Feature Interaction and Dependencies: Modeling Features for Re-engineering a Legacy Product Line. Proceedings of 2nd Software Product Line Conference (SPLC-2), San Diego, CA, USA, pp. 235-256.

6. Günter, A. (1995). Wissensbasiertes Konfigurieren. Infix, St. Augustin.

7. Günter, A. and Hotz, L. (1999). KONWERK – A Domain Independent Configuration Tool. Proceedings of Configuration (AAAI 1999 Workshop), Orlando, Florida, USA, pp. 10-19.

8. Hein, A., Schlick, M. and Vinga-Martins, R. (2000). Applying Feature Models in Industrial Settings. In Software product lines - Experience and research directions (Ed Donohoe P.). Kluwer Academic Publishers, pp. 47-70.

9. Hein, A., MacGregor, J. and Thiel S. (2001). Configuring Product Line Features. Proceedings of Feature Interaction in Composed Systems (ECOOP 2001 Workshop), Budapest, Hungary.

10. Hotz, L. and Vietze, T. (1995). Innovatives Konfigurieren in technischen Domänen. Proceedings of 9. Workshop Planen und Konfigurieren (PuK 1995). DFKI Saarbrücken, Kaiserslautern, Germany.

11. Hotz, L. and Krebs, T. (2003). Supporting the Product Derivation Process with a Knowledge-based Approach. Proceedings of Software Variability Management (ICSE 2003 Workshop), Portland, Oregon, USA.

12. Kang, K., Cohen, S., Hess, J., Novak, W. and Peterson, S. (1990). Feature-oriented Domain Analysis (FODA) – A Feasibility Study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon University, Pittsburgh, PA, USA.

13. Kang, K. and Lee, J. and Donhoe, P. (2002). Feature-oriented Product Line Engineering. IEEE Software, Vol. 7 (8), pp. 58-65.

14. Klein, M. and Noy, N. (2003). A Component-based Framework for Ontology Evolution. Technical Report IR-504, Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands.

15. Krebs, T. and Hotz, L. (2003). Needed Expressiveness for Representing Features and Customer Requirements. Proceedings of Modeling Variability for Object-Oriented Product Lines (ECOOP 2003 Workshop), Darmstadt, Germany.

16. Robak, S. and Franczyk, B. (2001). Feature Interaction and Composition Problems in Software Product Lines. Proceedings of Feature Interaction in Composed Systems (ECOOP 2001 Workshop), Budapest, Hungary.