Variability Modelling for Model-Driven Development of Software Product Lines*

Ina Schaefer
Dept. of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
schaefer@chalmers.se

***Abstract*—Model-driven development of software-intensive systems aims at designing systems by stepwise model refinement. In order to create software product lines by model-driven development, product variability has to be represented on every modelling level and preserved under model refinement. In this paper, we propose Δ-modelling as an generally applicable variability modelling concept that is orthogonal to model refinement. Products on each modelling level are represented by a core model and a set of Δ-models specifying changes to the core to incorporate product features. Core and Δ-models can be refined independently to obtain a more detailed model of the product line. Based on a formalization of Δ-modelling, we establish conditions that model refinement and model configuration commute resulting in an incremental model-driven development process.**

***Keywords*-Software Product Lines; Variability Modelling; Model-driven Development; Model Refinement**

I. Introduction

Model-driven development [1] of software-intensive systems aims at reducing design complexity by shifting the focus during system development from implementation to modelling. A model is an abstraction of a system with respect to certain system aspects. In model-driven development, an initial system model is successively refined by adding details relevant in particular design phases. A software product line [2], [3] is a set of systems with well-defined commonalities and variabilities. In order to use model-driven development in software product line engineering, the variability of the different products has to be represented within the used modelling concepts and preserved under model refinement.

The variability of products in software product lines is currently predominantly captured by feature models [4]. Features represent important product characteristics. A feature model determines a set of products by the set of valid feature configurations. However, features at the level of a feature model are merely labels [5]. Hence, feature-based variability has to be mapped to the modelling concepts used on each modelling level [6] in order to design a product line by a model-driven development process.

Existing approaches to integrate feature-based variability into modelling languages can be classified in two main directions [7]. First, negative variability-based approaches consider one model for all products of a product line that is augmented with variant annotations determining which model elements are present in which products [8], [9], [10], [11]. Second, positive variability-based approaches [6], [7], [12], [13], [14] associate model fragments to features and compose them for a given feature configuration. However, most approaches only focus on modelling concepts used on one modelling level and do not consider how the variability representation can be preserved under model refinement.

In order to define a seamless model-driven development process for software product lines, we propose Δ-modelling, a general concept integrating variability modelling with model refinement. On each modelling level, product line variability is represented by a core model and a set of Δ-models. The core model represents a valid product of the product line. Δ-models specify changes of the core model, i.e., additions, modifications and removals of model fragments, in order to capture further products. An application condition attached to a Δ-model determines for which feature configuration a Δ-model is applicable. A product model for a feature configuration is obtained by applying the modifications specified by the Δ-models with valid application conditions.

For refinement, the core model and every Δ-model are transformed independently into a more detailed core model or Δ-model, respectively. The internal structure of the core and Δ-models, as well as the application conditions of the Δ-models are preserved. If the specified modifications in the refined Δ-models satisfy local refinement compatibility conditions, a refined product model for a feature configuration can be obtained in two ways: first, the product model is configured on the higher level of abstraction and afterwards transformed to a refined model; or second, the core and Δ-models are refined and afterwards configured by applying the modifications of the refined Δ-models to the refined core model. The commutativity of model refinement and model configuration builds the basis for incremental model-driven development of software product lines.

The Δ-modelling concept provides an integrated variability modelling approach for model-driven development of software product lines. Its main characteristics are:

- Δ-modelling is independent of a concrete modelling or implementation language. It can be instantiated to concrete modelling or implementation languages by defining the semantics of Δ-application.

*This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) and by the European project HATS, funded in the Seventh Framework Program.

- Combinations of features can be explicitly captured by flexible application conditions attached to Δ-models.
- Modular and evolutionary system development is facilitated by adding Δ-models to an existing model.
- Model refinement is orthogonal to variability modelling. Core and Δ-models are refined independently such that variability is expressed by the same structural concepts on all modelling levels.
- The commutativity of model configuration by Δ-application and model refinement provides the basis for an incremental development process by stepwise refinement of core and Δ-models.

The outline of this paper is as follows: In Section II, we review related work. In Section III, we explain the Δ-modelling approach at an example of a trading system product line. In Section IV, we formalize Δ-modelling and extend this formalization to model refinement in Section V. SectionVI concludes with an outlook to future work.

II. Related Work

Model-driven engineering for software product line development is proposed in [10], [15] in order to resolve product variability by model transformations. A model in the problem domain, usually a feature model, is transformed into a model in the solution domain, e.g., a product model [7], product architecture [16] or product implementation [17].

The existing approaches to represent feature-based variability can be classified into two main directions [7]. Annotative approaches specify negative variability. They consider one model representing all products of a product line. Variant annotations, e.g., using UML stereotypes [8], [9], [10], [11], define which parts of the model have to be removed to derive the model of a concrete product. [5] associates presence conditions to modelling elements to be removed in certain feature configurations.

Compositional approaches capture positive variability. Model fragments are associated with features and composed for a particular feature configuration. A prominent example is the AHEAD [12] approach. A product is built by stepwise refinement of a base module with a sequence of feature modules. In [6], [7], [13], models are constructed by aspect-oriented modelling techniques. [14] applies model superposition to compose model fragments. In [18], a product model is obtained by composition and refined by model transformation. [19] propose to represent model variability by a base model and associated variability and resolution models determining how modelling elements of the base model have to be replaced for a particular product model. The base model is similar to the core model in the Δ-modelling approach while variability and resolution models correspond to Δ-models, but are not directly connected product features.

Most of the above approaches only focus on the representation of variability on a single modelling layer. In [9], different modelling levels during system development are considered, but variability resolution is based on textual decision models that are separated from the system models. In contrast, Δ-modelling facilitates a seamless representation of variability inbetween different modelling layers.

The notion of program deltas is introduced in [20] to describe the modification of an object-oriented program, e.g., by introduction of new fields or extension of methods. The mapping of collaborative features to models in [6] is similar to Δ-models. Collaborative features can modify a core model by additions, removals and modifications, but require a one-to-one relationship to a feature. In [21], Δ-modelling is presented as an approach to develop product line artifacts suitable for automated product derivation.

Feature-oriented model-driven development (FOMDD) [22] combines feature-oriented programming (FOP) with model-driven engineering. In FOMDD, a product can, first, be composed from a base module and a sequence of feature modules and afterwards transformed to another product model. Second, the base module and the feature modules can be transformed and then composed to a transformed product model. This is similar to the commutativity between model refinement and model configuration in Δ-modelling. FOP can be seen as a special case of Δ-modelling. Feature modules are always associated to exactly one feature, whereas Δ-models explicitly consider combinations of features. In feature modules, only additions and modifications can be specified. In contrast, Δ-models may contain removals of model parts. While the base module in FOP is fixed by the mandatory features, in Δ-modelling, any valid product can be chosen as core model enabling a flexible product line design.

III. Variability Modelling using Δ-Models

Figure 1 shows an overview of the model-based development process for software product lines using the Δ-modelling approach. First, an initial model of the product line is created that captures the variability of the feature model. From this initial model on a high level of abstraction, successively refined models are constructed that describe more detailed aspects of the considered products.

On each modelling level, product variability is captured by a core model and a set of Δ-models. A core model corresponds to a valid product of the product line. Δ-models specify changes to the core model by additions, modifications and removals of model fragments in order to represent further products. An application condition is attached to every Δ-model determining for which feature configurations the specified changes are to be carried out. In order to obtain a product model for a feature configuration, the changes specified by Δ-models with valid application condition are applied to the core. The concept of Δ-modelling to express variability is independent of a concrete modelling language. The modelling constructs used on each modelling level can be chosen to appropriately represent the considered system aspects. The application conditions attached to the Δ-models create

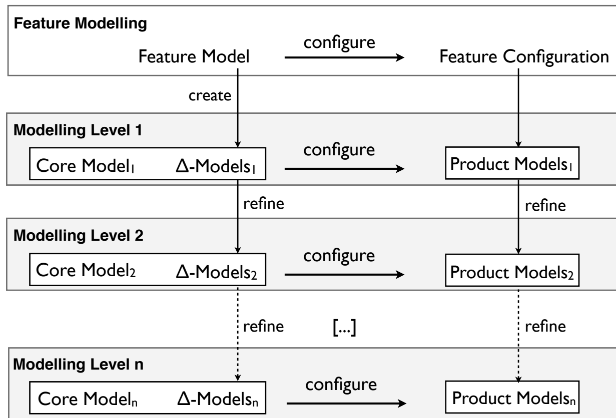

Figure 1. Model-driven development with Δ-Modelling

the connection between the features in the feature model and product variability on the different modelling levels. If the selection of a feature influences the choice of the modelling language, e.g., if a feature refers to the used implementation framework, core and the Δ-models can be seen tuples containing the specifications of the core and Δ-models in the respective modelling formalisms, while the general variability structure is preserved.

The step from a feature model of a product line to the initial core model and the set of Δ-models is a creative process, since product line variability can in general be represented in different ways. The variability structure provided by the initial modelling level provides the variability structure of the lower, more refined modelling levels (cf. Figure 1). A core model is refined to a more detailed core model. Δ-models are refined to more detailed Δ-models with the same application condition. An important property of the refinement between two modelling levels is that it commutes with model configuration by Δ-application. This means that a refined configured product model can be obtained in the following two ways. First, the product model for a feature configuration is configured from the core model and the applicable Δ-models and afterwards refined. Second, the core model and the Δ-models are refined, such that afterwards the refined product can be configured. This commutativity property provides the basis for an incremental model-based development process by stepwise model refinement.

Example We illustrate variability modelling based on Δ-models at the case example of a software product line of trading systems. The Common Component Modeling Example (CoCoME) [23] describes a software system handling payment transactions in supermarkets. It was extended to a software product line in [24]. The variability of the products are expressed in the feature diagram [4] shown in Figure 2. Mandatory features are represented by a filled circle, optional features with an empty circle. Alternative features are specified with a filled triangle if at least one feature has to be selected or by an empty triangle if exactly one features has to be selected. Constraints between features are represented by explicit links. A

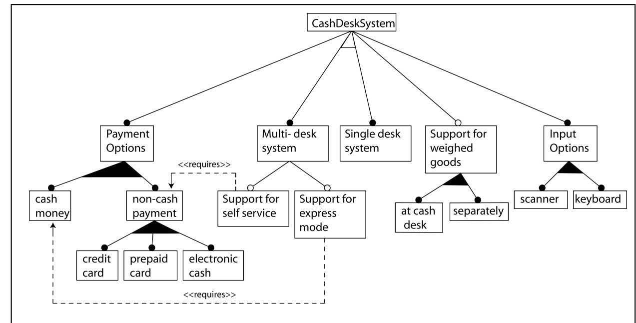

Figure 2. Feature Model for the CoCoME Software Product Line

product in the trading system product line has different payment options, i.e., cash payment or payment by credit card, prepaid card or electronic cash. At least one payment option has to be chosen for a valid configuration. Product information can be entered using a keyboard or a scanner, where at least one option has to be selected. Furthermore, the system has optional support to weigh goods, either at the cash desks or at separate facilities. A trading system can be configured as a single-desk system with only one cashier or as a multi-desk system with several of cashiers. A multi-desk system can optionally comprise an express mode which requires cash payment or a self-service mode requiring non-cash payment.

A core model represents a product for a valid feature configuration. Thus, it can be developed by well-established single application engineering techniques as a standard product model. In the example, the feature configuration containing the keyboard, the cash payment and the single-desk system features is selected as core configuration.

Component Modelling Level In our example, we start the model-based development process at the component level by representing the core model by a UML component diagram [25] and its variability by component diagram Δ-models that are an extension of UML component diagrams with annotations for the specified changes. In a second step, the component diagrams are refined to UML class diagrams showing in more detail how the components are implemented. Figure 3 depicts the component diagram specifying the core product of the trading system product line with the keyboard, cash payment and single-desk system features. It contains a Cash Desk component dealing with cash payment and an Inventory component keeping the store inventory. Every time a product is entered at the cash desk, the price of the product is requested from the inventory.

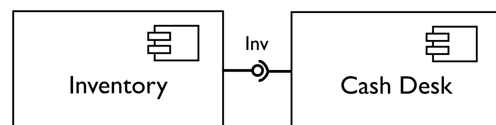

Figure 3. Core Component Diagram

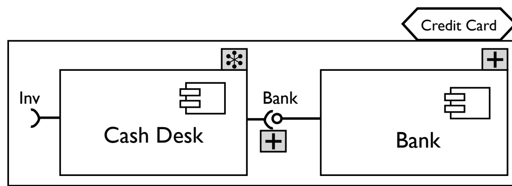

Figure 4. Component Diagram Δ-Model

Figure 4 depicts the component diagram Δ-model containing the modifications of core component diagram to include credit card functionality. A Bank component has to be added specified by the + annotation at the Bank component. Additionally, the Cash Desk component has to be modified to handle credit card payment which indicated by $*$ annotation at the Cash Desk component. In order to realize the communication with the Bank component, an required interface and a corresponding connection to the bank component have to be added. The application condition of this Δ-model (in the top right hand corner) defines that the modifications are carried out if the credit card feature is selected. In Figure 5, the component diagram for a single-desk system containing keyboard input, cash payment and credit card payment is depicted that results from applying the Δ-model for credit card payment (cf. Figure 4) to the core component diagram (cf. Figure 3).

Class Modelling Level Each component of the trading system product line can be refined to a class diagram. The class diagram represents the internal component structure and can be used as basis for an implementation. The interactions between the components are not considered on the class diagram level because they are already captured on the component modelling level. In the Δ-modelling approach, core and Δ-models are refined independently. A refined product model is obtained by applying the refined Δ-models to the refined core model.

Figure 6 shows the class diagram for the Cash Desk component contained in the core. It comprises a Cash Desk class implementing the main functionality of the cash desk, a Keyboard class handling the input from the keyboard and a Display class providing output to a display. Figure 7 depicts the class diagram Δ-model specifying the modifications of the core class diagram Cash Desk component to incorporate the credit card feature. In order to provide credit card payment functionality, a Card Reader class is required. The Cash Desk class is modified by adding a reference to the bank, by adding methods to deal with the credit card payment and by modifying the existing payment methods.

Figure 5. Product Component Diagram

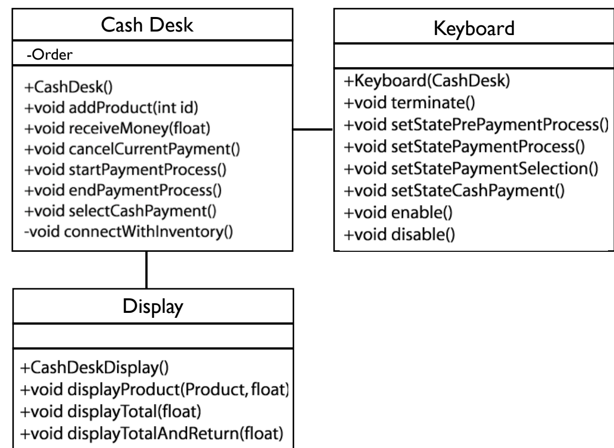

Figure 6. Core Class Diagram

Model Refinement and Configuration The class diagram for the configured Cash Desk component with the basic features and the credit card feature can be obtained in two different ways. First, the core model and the Δ-model on the component modelling level (cf. Figure 3 and 4) can refined to a core and Δ-model on the class diagram level (cf. Figures 6 and 7) and configured to a class diagram by the standard configuration procedure. Second, a component diagram including the Cash Desk component with the basic features and the credit card feature can be configured on the component diagram level (cf. Figure 5). Afterwards, the configured Cash Desk component can be refined to a class diagram specifying the component's structure in more detail.

Model refinement and model configuration commute in the example because the refinement of the component diagram Δ-models is compatible with model configuration. Compatibility requires that for a component added (or removed) by a component diagram Δ-model, in the class diagram refinement of this Δ-model, all parts of the class diagram are specified as added (or removed) as well. If a component is modified in a component diagram Δ-model, in the refined class diagram, the parts of the class diagram

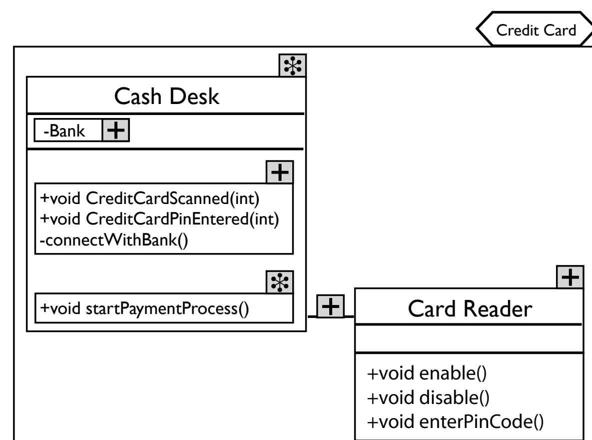

Figure 7. Class Diagram Δ-Model

may be specified as added, modified or removed. However, the change operations resulting from refinement of a modification operation have to satisfy a local refinement compatibility condition. This condition requires that the class diagram obtained by applying the refined component Δ-model to the class diagram of a refined core component is the same as the class diagram refinement of the same component configured on the component diagram level.

IV. Formalizing Δ-Modelling

Variability modelling using Δ-models is a general approach that is not limited to specific modelling concepts, such as component or class diagrams used in Section III. Δ-modelling can be applied to any modelling or implementation language by defining the semantics of the change operations specified in the Δ-models for the concrete language. The number of modelling layers also depends on the concrete application and is not limited by the Δ-modelling approach. For instance, in Section III, use case diagrams separated into core and Δ-diagrams could be used to represent the requirements of the set of systems under development and subsequently be refined to component diagrams.

In order to show the general applicability of Δ-modelling, we base the following formalization on a general notion of models. A *model* contains a set of modelling elements E that can, for instance, represent components or classes (by their names). The set of modelling elements E describes the (domain-specific) concepts used in the models. Furthermore, a model contains relations between modelling elements representing correspondences, such as connections between required and provided interfaces in component diagrams. For simplicity, we restrict our notion of a model to contain only one binary relation R over the set of modelling elements E. This allows us to consider relations formally while keeping the model and its formal treatment simple. In a concrete model instantiation, a set of relations can be defined to express different relationships between elements.

Definition 1 (Models): Let E be a set of modelling elements. A *model* M is a tuple $M = (E, R)$ where $R \subseteq E \times E$ is a relation over the modelling elements.

A *core model* represents a product for a valid feature configuration. This allows treating the core model in the same way as any product model. We define the set of valid feature configurations as a subset of the powerset of the set of features.

Definition 2 (Core and Product Models): For a set of features $F = \{f_1, \dots f_n\}$, let $\mathcal{F} \subseteq \mathcal{P}(F)$ denote the set of valid feature configurations. A core model (product model) is a triple $C = (E, R, f)$ where $f \in \mathcal{F}$ is a valid feature configuration, and (E, R) is a model representing the feature configuration f.

A Δ-*model* specifies changes to a core model to model other products. For a core model $C = (E, R, f)$, a Δ-model defines additions, modifications and removals of modelling elements $e \in E$, and additions and removals of tuples in the relation R. A Δ-model has an *application condition* determining under which feature configurations the specified changes have to be carried out. Application conditions are logical (e.g., Boolean) constraints over the features contained in the feature model. A Δ-model does not necessarily refer to exactly one feature, but potentially to a combination of features. This allows very flexible Δ-models as combinations of features can be handled individually. For example, if a feature model contains two features A and B, the Boolean constraint $(A \wedge \neg B)$ denotes that the modifications are only carried out for a feature configuration if feature A is selected and feature B is not selected. The granularity of the application conditions determines the number of Δ-models that have to be created to ensure that all features present in the feature model are appropriately captured.

Definition 3 (Δ-Models): A Δ-model over a model $M = (E, R)$ is a tuple $\Delta = (\varphi, Op)$ where the application condition φ is a constraint over the set of features $F = \{f_1, \dots f_n\}$ and $Op = \{op_1, \dots, op_m\}$ is a set of modification operations over the model M with

$$op_i ::= \texttt{add}\, e \mid \texttt{mod}\, e \mid \texttt{rem}\, e \mid \texttt{add}\, r(e_1, e_2) \mid \texttt{rem}\, r(e_1, e_2)$$

In order to obtain a product model for a feature configuration $f \in \mathcal{F}$, all Δ-models with valid application condition for the feature configuration f are applied to the core model. This can involve different Δ-models that are applicable for the same feature. To limit the occurrence of conflicts between changes targeting the same modelling elements and relations, first all additions, then all modifications and finally all removals are performed. In order to express this ordering formally, we assume that Δ-models are *normalized*, i.e., their change operations contain only additions, only modifications, or only removals. A Δ-model $\Delta = (\varphi, Op)$ can be normalized by splitting it into three disjoint normalized Δ-models $\Delta_a = (\varphi, Op_a)$, $\Delta_m = (\varphi, Op_m)$ and $\Delta_r = (\varphi, Op_r)$ such that $Op = Op_a \uplus Op_m \uplus Op_r$. We call a set of Δ-models $\Delta = \{\Delta_1, \dots, \Delta_n\}$ *sorted* if and only if there exist i, j with $1 \leq i \leq j \leq n$, such that $\Delta_1, \dots, \Delta_i$ contain only additions, $\Delta_{i+1}, \dots, \Delta_j$ contain only modifications, and $\Delta_{j+1}, \dots, \Delta_n$ contain only removals. The operation $\nu(\Delta)$ transforms a set of Δ-models into a set of normalized and sorted Δ-models. The application function $apply(M, Op)$ modifying a model M by the change operations Op is defined in Definition 5.

Definition 4 (Configuration): Let $C = (E, R, f_c)$ be a core model and $\Delta = \{\Delta_1, \dots, \Delta_n\}$ be a sequence of normalized and sorted Δ-models with $\Delta_i = (\varphi_i, Op_i)$ for all i. For a feature configuration $f \in \mathcal{F}$, a product model $P_f = (E_P, R_P, f)$ is configured by $conf((E, R), \{\Delta_1, \dots, \Delta_n\}, f)$ where for a model $M = (E_M, R_M)$, its configuration is defined by $conf(M, \{\Delta_1, \dots, \Delta_n\}, f) =$

$$\begin{cases} conf(apply(M, Op_1), \{\Delta_2, \dots, \Delta_n\}, f) & \text{if } f \models \varphi_1 \\ conf(M, \{\Delta_2, \dots, \Delta_n\}, f) & \text{otherwise} \end{cases}$$

and $conf(M, \emptyset, f) = (E_M, R_M, f)$.

The ordering in which the change operations specified in a single Δ-model are applied to a model is not fixed, which can also be seen as simultaneous application of the specified changes. The same modelling element can be added several times, but only occurs once in the model. Similarly, a tuple in the relation is added only once, if the related modelling elements are contained in the model. The modification of a modelling element also causes that the element is replaced in all relational tuples in which the original modelling element is contained. If an element is removed, also all relational tuples containing this element are removed from the relation. The application of a change operation is undefined if a modelling element e is modified that is not contained in the core or added before by another Δ-model, or if a modelling element or a relational tuple is removed, that is not contained in the core or added before by another Δ-model.

Definition 5 (Δ-Application): The application of a set of change operations $Op = \{op_1, ..., op_n\}$ to a model $M = (E, R)$ is defined by the application function $apply$:

- $apply(M, \emptyset) = M$
- $apply(M, Op) = apply(apply(M, op_i), Op \setminus \{op_i\})$
- $apply(M, \texttt{add } e) = (E \cup \{e\}, R)$
- $apply(M, \texttt{mod } e) = (E \setminus \{e\} \cup \{e'\}, R')$, if $e \in E$ where $e' \in E$ is the result of the modification of $e \in E$ and $R' = \{(e_1, e_2) \mid (e_1, e_2) \in R, e_1, e_2 \neq e\} \cup \{(e', e_2) \mid (e, e_2) \in R, e_2 \neq e\} \cup \{(e_2, e') \mid (e_2, e) \in R, e_2 \neq e\} \cup \{(e', e') \mid (e, e) \in R\}$
- $apply(M, \texttt{rem } e) = (E \setminus \{e\}, R')$, if $e \in E$ where $R' = \{(e_1, e_2) \mid (e_1, e_2) \in R \wedge e_1, e_2 \neq e\}$
- $apply(M, \texttt{add } r(e1, e2)) = (E, R \cup \{r(e_1, e_2)\})$, if $e_1, e_2 \in E$
- $apply(M, \texttt{rem } r(e1, e2)) = (E, R \setminus \{r(e_1, e_2)\})$, if $e_1, e_2 \in E$ and $r(e_1, r_2) \in R$.

Despite using normalized and sorted Δ-models during configuration, there can still be conflicts between the change operations specified in different Δ-models. A conflict occurs if a modelling element or tuple in a relation is added and removed by two different Δ-models, if a modelling element is modified and removed by two different Δ-models or if a modelling element is modified by two different Δ-models. This indicates that the granularity of the Δ-models and their application conditions is too coarse. Conflicts can be removed by splitting Δ-models and refining them to explicitly cover the conflicting feature combinations.

Definition 6 (Conflicts in Δ-Models): A set of Δ-models $\Delta = \{\Delta_1, \ldots, \Delta_n\}$ contains a conflict if for a feature configuration $f \in \mathcal{F}$, there are Δ-models Δ_i and Δ_j with $i \neq j$, $f \models \varphi_i$ and $f \models \varphi_j$, and there exists $e \in E$, or $e_1, e_2 \in E$ and $r(e_1, e_2) \in R$, such that one of the following holds:

- $\texttt{add } e \in Op_i$ and $\texttt{rem } e \in Op_j$
- $\texttt{add } r(e_1, e_2) \in Op_i$ and $\texttt{rem } r(e_1, e_2) \in Op_j$
- $\texttt{mod } e \in Op_i$ and $\texttt{rem } e \in Op_j$
- $\texttt{mod } e \in Op_i$ and $\texttt{mod } e \in Op_j$

A core model $C = (E, R, f_b)$ and a set of Δ-models Δ are *well-defined* if for all valid feature configurations $f \in \mathcal{F}$, all applications of Δ-operations are defined and there are no conflicts between any two Δ-models. Well-definedness is a prerequisite for commutativity of model configuration and model refinement.

V. Model Refinement and Configuration

Based on the formalization of the Δ-modelling approach in Section IV, model refinement of core and Δ-models can be defined. A model is transformed to a more detailed model by refining the contained modelling elements to models themselves, as in the example in Section III, components are refined to class diagrams showing their internal structure. Relations between modelling elements are not considered for refinement, such as connections between components are only relevant on the component modelling level.

Definition 7 (Model Refinement): The refinement operation $refine$ maps every modelling element $e \in E$ to a model M such that $refine(e) = M_e = (E_e, R_e)$. The refinement M' of a model $M = (E, R)$ is defined by

$$refine(M) = (\{M'' \mid M'' = refine(e), e \in E\}, \{r(refine(e_1), refine(e_2)) \mid r(e_1, e_2) \in R\})$$

The core model and all other product models can be refined using the above definition of model refinement. In order to refine Δ-models, the specified addition, modification and removal operations on modelling elements and relations have to be refined. The addition of a modelling element is refined to a set of addition operations for the elements and the relational tuples of the model obtained by refining the modelling element. The removal of a modelling element is refined to a set of remove operations for the modelling elements and relational tuples of the model resulting from refining the modelling element. The modification of a modelling element is refined to a set of addition, modification and removal operations for modelling elements and relational tuples obtained by refining the modelling element. Change operations for relations are removed during Δ-refinement. The application condition of a Δ-model remains unchanged such that the variability structure is preserved. In the example in Section III, the component diagram Δ-model (cf. Figure 4) is refined to a class diagram Δ-model (cf. Figure 7) according to the following definition.

Definition 8 (Δ-Refinement): The refinement of a Δ-model $\Delta = (\varphi, Op)$ is defined by $refine(\Delta) = (\varphi, refine(Op))$ where $refine(\{op_1, ..., op_n\}) = \{refine(op_1), ..., refine(op_n)\}$ and

- $refine(\texttt{add } e) = \{\texttt{add } e' \mid e' \in E_e\} \cup \{\texttt{add } r(e'_1, e'_2) \mid r(e'_1, e'_2) \in R_e, e'_1, e'_2 \in E_e\}$
- $refine(\texttt{rem } e) = \{\texttt{rem } e' \mid e' \in E_e\} \cup \{\texttt{rem } r(e'_1, e'_2) \mid r(e'_1, e'_2) \in R_e, e'_1, e'_2 \in E_e\}$
- $refine(\texttt{mod } e) = \{\texttt{op } e' \mid e' \in E_e\} \cup \{\texttt{op } r(e'_1, e'_2) \mid r(e'_1, e'_2) \in R_e, e'_1, e'_2 \in E_e\}$ where $\texttt{op} \in \{\texttt{add}, \texttt{rem}, \texttt{mod}\}$
- $refine(\texttt{add } r(e_1, e_2)) = refine(\texttt{rem } r(e_1, e_2)) = \emptyset$

The configuration of a refined model for a feature configuration $f \in \mathcal{F}$ is performed in three steps. First, the original model on the higher abstraction level is configured subject to the feature configuration f. Second, for every modelling element e included in the resulting product model, the refined core model restricted to the modelling element e is configured using the refined Δ-models restricted to the modelling element e subject to the feature configuration f. If a modelling element e is not contained in the core model, but introduced by a Δ-model, the refined core model restricted to the modelling element e is an empty model. Third, the refined modelling elements replace their non-refined version in the configured original model. The result is a model that contains models as modelling elements and relations between these models. In the example in Section III, a configured, refined model is a component diagram in which the components contain class diagrams showing their detailed internal structure. The restriction of a core model $C = (E, R, f_b)$ to a modelling element $e \in E$ is defined by $C|_e = (\{e\}, \emptyset)$ if $e \in E$ and by $C|_e = (\emptyset, \emptyset)$, otherwise. Further, we define $\Delta|_e = \{\Delta_1|_e, \ldots, \Delta_n|_e\}$ as the set of Δ-models only modifying element $e \in E$ where $\Delta_i|_e = (\varphi_i, \{\texttt{op}\ e \in Op_i\})$ for $\texttt{op} \in \{\texttt{add}, \texttt{rem}, \texttt{mod}\}$.

Definition 9 (Configuration of Refined Models): Let $C = (E, R, f_c)$ be a core model, $\Delta = \{\Delta_1, \ldots, \Delta_n\}$ a set of normalized and sorted Δ-models and $f \in \mathcal{F}$ a feature configuration. Let $P_f = (E_P, R_P, f) = \mathit{conf}((E, R), \{\Delta_1, \ldots, \Delta_n\}, f)$ be the configured original model for the feature configuration f. Further, let $M_e = \mathit{conf}(\mathit{refine}(C|_e), \nu(\mathit{refine}(\Delta|_e)), f)$ be the refined configured modelling element for $e \in E_P$.

The refined configured model is defined by $P_r = \mathit{conf}_{\mathit{ref}}(\mathit{refine}(E, R), \mathit{refine}(\Delta), f)$ with $P_r =$

$$(\{M_e \mid e \in E_P\}, \{r(M_{e_1}, M_{e_2}) \mid r(e_1, e_2) \in R_P\}, f)$$

The commutativity of model refinement and model configuration by Δ-application constitutes the basis for the incremental model-based development of software product lines by stepwise refinement of core and Δ-models. The requirement for commutativity is that the refinement of the change operations specified in Δ-models is compatible with model configuration. For addition and removal operations, compatibility is ensured by the definition of Δ-model refinement in Definition 8. For the refinement of the modification operations, a local refinement compatibility condition has to be established. This local refinement compatibility condition requires that the result of applying the refined modification operation to the refined modelling element in core model is the same as the refinement of the modelling element that has been configured on the non-refined modelling level.

Definition 10 (Refinement Compatibility): Let $e' \in E$ be the result of applying the modification operation $\texttt{mod}\ e$ to the modelling element $e \in E$. The local refinement compatibility constraint for the operation $\texttt{mod}\ e$ holds iff

$$\mathit{apply}(\mathit{refine}(e), \mathit{refine}(\texttt{mod}\ e)) = \mathit{refine}(e')$$

The following theorem states that model refinement and model configuration commute if all modification operations satisfy refinement compatibility.

Theorem 1 (Commutativity): For a feature configuration $f \in \mathcal{F}$, a core model $C = (E, R, f_c)$ and a set of well-defined Δ-models $\Delta = \{\Delta_1, \ldots, \Delta_n\}$ and a refinement *refine* on the modelling elements $e \in E$, if all modification operations $\texttt{mod}\ e \in Op_i$ satisfy the refinement compatibility condition, then it holds that $\mathit{refine}(\mathit{conf}((E, R), \Delta, f)) = \mathit{conf}_{\mathit{ref}}(\mathit{refine}(E, R), \mathit{refine}(\Delta), f)$.

Proof: By induction on the set of Δ-models and a case distinction on their `add`, `mod`, `rem` operations. For `add` and `rem` operations, the definition of Δ-refinement in Definition 8 is used. For `mod` operations, the refinement compatibility condition from Definition 10 is assumed. ■

Commutativity of model refinement and model configuration provides that basis for an incremental model-driven development process for software product lines. After the initial core and Δ-models have been created to capture the variability of the feature model, this initial variability structure is preserved on each modelling level by the independent refinement of core and Δ-models.

VI. Conclusion

Δ-modelling is an variability modelling approach for model-driven development of software product lines. Product variability is expressed by core and Δ-models on all modelling levels that are preserved under model refinement. For a concrete development process, the semantics of Δ-application has to be defined for the language concepts used on each modelling level. Model configuration by Δ-application can be automated, e.g., by aspect-oriented model weaving techniques [6], [7], [13] or model superimposition [14]. In [21], the Δ-modelling approach has been implemented using frame technology.

For future work, we aim at providing tool support and guidelines how to develop initial core and Δ-models for a given feature model. This will be complemented by modular analyses establishing conflict-freedom and well-definedness of core and Δ-models using existing variability analysis techniques based on confluence analysis. Besides, we want to extend Δ-modelling with explicit conflict resolution by imposing a partial order between Δ-models in order to avoid the normalization of Δ-models during configuration.

Acknowledgments The author wants to thank Arnd Poetzsch-Heffter and Alexander Worret for initial collaboration on the subject of this work.

References

[1] B. Selic, "The Pragmatics of Model-driven Development," *IEEE Software*, Sept 2003.

[2] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.

[3] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, Heidelberg, 2005.

[4] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie Mellon Software Engineering Institute, Tech. Rep., 1990.

[5] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," in *GPCE*, 2005.

[6] F. Heidenreich and C. Wende, "Bridging the Gap Between Features and Models," in *Aspect-Oriented Product Line Engineering (AOPLE'07)*, 2007.

[7] M. Völter and I. Groher, "Product Line Implementation using Aspect-Oriented and Model-Driven Software Development," in *SPLC*, 2007, pp. 233–242.

[8] T. Ziadi, L. Hélouët, and J.-M. Jézéquel, "Towards a UML Profile for Software Product Lines," in *Workshop on Product Familiy Engineering (PFE)*, 2003, pp. 129–139.

[9] C. Atkinson, J. Bayer, , and D. Muthig, "Component-Based Product Line Development: The KobrA Approach," in *SPLC*, 2000.

[10] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, and A. Solberg, "An MDA-based framework for model-driven product derivation," in *Software Engineering and Applications (SEA)*, 2004.

[11] H. Gomaa, *Designing Software Product Lines with UML*. Addison Wesley, 2004.

[12] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Trans. Software Eng.*, vol. 30, no. 6, 2004.

[13] N. Noda and T. Kishi, "Aspect-Oriented Modeling for Variability Management," in *SPLC*, 2008.

[14] S. T. Sven Apel, Florian Janda and C. Kästner, " Model Superimposition in Software Product Lines," in *International Conference on Model Transformation (ICMT)*, 2009.

[15] S. Deelstra, M. Sinnema, J. van Gurp, and J. Bosch, "Model Driven Architecture as Approach to Manage Variability in Software Product Families," in *Workshop on Model Driven Architecture: Foundations and Applications (MDAFA 2003)*, 2003.

[16] G. Botterweck, L. O'Brien, and S. Thiel, "Model-driven Derivation of Product Architectures," in *Automated Software Engineering (ASE)*, 2007, pp. 469–472.

[17] G. Botterweck, K. Lee, and S. Thiel, "Automating Product Derivation in Software Product Line Engineering," in *Software Engineering*, 2009, pp. 177–182.

[18] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel, "Reconciling Automation and Flexibility in Product Derivation," in *SPLC*, 2008.

[19] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen, "Adding Standardized Variability to Domain Specific Languages," in *SPLC*, 2008.

[20] R. Lopez-Herrejon, D. Batory, and W. Cook, "Evaluating Support for Features in Advanced Modularization Technologies," in *ECOOP*, 2005.

[21] I. Schaefer, A. Worret, and A. Poetzsch-Heffter, "A Model-Based Framework for Automated Product Derivation," in *Model-driven Approaches in Software Product Line Engineering (MAPLE 2009)*, 2009.

[22] S.Trujillo, D. Batory, and O. Díaz, "Feature Oriented Model Driven Development: A Case Study for Portlets," in *ICSE*, 2007.

[23] S. Herold *et al.*, "CoCoME - The Common Component Modeling Example," in *Common Component Modeling Example*, A. Rausch *et al.*, Eds. Springer-Verlag, 2008, pp. 16 – 53.

[24] A. Worret, "Automated Product Derivation for the CoCoME Software Product Line: From Feature Models to CoBoxes," Master's thesis, University of Kaiserslautern, March 2009.

[25] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology, 2004.