A Model-Based Framework for Automated Product Derivation*

Ina Schaefer, Alexander Worret, Arnd Poetzsch-Heffter
Software Technology Group
TU Kaiserslautern
Kaiserslautern, Germany
{*inschaef* | *worret* | *poetzsch*}*@cs.uni-kl.de*

***Abstract*—Software product line engineering aims at developing a set of systems with well-defined commonalities and variabilities by managed reuse. This requires high up-front investment for creating reusable artifacts, which should be balanced by cost reductions for building individual products. We present a model-based framework for automated product derivation to facilitate the automatic generation of products. In this framework, a model-based design layer bridges the gap between feature models and implementation artifacts. The design layer captures product line variability by a core design and Δ-designs specifying modifications to the core for representing product features. This structure is mapped to the implementation layer guiding the development of code artifacts capable of automatic product derivation. We evaluate the framework for a CoBox-based product line implementation using extended UML class diagrams for the design and frame technology for the implementation layer.**

***Keywords*-Software Product Lines; Automated Product Derivation; Model-based Development; Frame Technology**

I. Introduction

A *software product line* is a set of software systems with well-defined commonalities and variabilities [1]. Software product line engineering aims at developing these systems by managed reuse in order to reduce time to market and to increase product quality. The creation of reusable artifacts requires a high up-front investment which should be balanced by cost reductions for building individual products. Currently, derivation of single products requires manual intervention during application engineering, especially for product implementation, which can be tedious and error-prone [2]. Hence, it cannot be guaranteed that the overall development costs are reduced by product line engineering when compared to other reuse approaches.

Automated product derivation (or software mass customization [3]) is an approach to create single products by removing the need for manual intervention during application engineering. Besides, automated product derivation allows centralized product line maintenance and product line evolution, because modifications of the artifacts can automatically be propagated to the products. In order to be able to create products automatically, product line variability is restricted to configurative variability [4]. The different product configurations are captured in a feature model where features are designated product characteristics. Automated product derivation means that a product implementation for a particular feature configuration is automatically generated from the reusable product line artifacts. Software product line engineering processes, such as PuLSE [5] or KobrA [6], focus on managing product line variability in all software development phases, but leave product derivation as a manual activity. In [7], only organizational and technical requirements for automated product derivation are considered. Some approaches [8], [9] aim at automatically deriving design documents. However, no approach provides guidance for the design and implementation of product line artifacts capable of automated product derivation.

To overcome this problem, we propose a model-based framework for automated product derivation. A design layer bridges the gap between feature models and product implementations. During domain engineering, it guides the development of implementation artifacts capable of automated product derivation. On the design layer, a product line is described by a core design and a set of Δ-designs. The core design represents a product with a basic set of features. The Δ-designs define modifications to the core design that are necessary to incorporate specific product characteristics. Δ-designs can cover combinations of features. This makes the presented approach very flexible because modifications caused by several features can be designed differently from modifications caused by one of these features. In order to obtain a design for a product with a particular feature configuration during application engineering, the modifications specified by the respective Δ-designs are applied to the core. A design can be validated and verified before code artifacts are developed. Furthermore, designs can be refined based on the principles of model-driven development [10]. Refinements are orthogonal to product line variability because they can be performed in both core and Δ-designs equally. Therefore, the proposed approach serves as a basis for model-driven development of software product lines with automated product derivation.

In order to develop reusable code artifacts capable of automated product derivation, the structure of the design layer is mapped to the implementation layer. A product

*This work has been partially supported by the Rheinland-Pfalz Research Center for Mathematical and Computational Modelling $(CM)^2$ and by the European project HATS, funded in the Seventh Framework Program.

line implementation consists of a core implementation of the product described by the core design and a set of Δ-implementations corresponding to the Δ-designs which specify the modifications to the core implementation to realize the designated product characteristics. The core design and core implementation refer to a complete product and can be developed by single application engineering techniques. The implementation of a product for a particular feature configuration is obtained automatically during application engineering by applying the modifications of the respective Δ-implementations to the core implementation. The design layer is independent of a specific implementation technique. The only requirement for a concrete implementation technique is that the modifications of the core can be represented appropriately and applied automatically. The separation of design and implementation artifacts into core and Δ-designs/implementations allows a stepwise development of the software product line. The approach can easily deal with evolving software product lines by capturing new features in additional Δ-designs/implementations.

We have evaluated the proposed model-based framework at the development of a shopping system product line. In order to consider variable deployments, the implementation layer is based on the CoBox component model [11]. We developed a notation for CoBox-based core and Δ-designs. For implementing the product line variability, we applied frame technology [12]. The core implementation is captured by core frames and the Δ-implementations by sets of Δ-frames specifying the modifications to the core implementation.

The main advantages of the model-based framework for automated product derivation are:

- The separation of core and Δ-designs/implementations allows an evolutionary development of product lines.
- Product variability can be handled very flexibly because Δ-designs/implementations allow representing modifications caused by combinations of features.
- The design layer facilitates model-based validation and verification before implementation.
- The framework can be used with different implementation techniques to exploit their strengths in particular application domains.
- The framework serves a basis for model-driven development of software product lines with automated product derivation because refinements are orthogonal to product line variability.

This paper is organized as follows: In Section II, we review related work. In Section III, we present our model-based framework for automated product derivation that is realized in Section IV and evaluated in Section V. Section VI concludes the paper with an outlook to future work.

II. Related work

Model-driven development [10] is increasingly used in software product line engineering. Many approaches focus on modeling product line variability. In KobrA [6], UML diagrams are annotated with variant stereotypes to describe variation points in models. In [13], a UML profile for representing product line variability is introduced. However, resolving the modeled variabilities requires additional documents and manual intervention.

In [14], [15], the general idea to use model-driven development for product derivation is advocated. Models in the problem domain, which correspond to feature models of product lines, are stepwise transformed to models in a solution domain, i.e. models of products or product implementations. However, these approaches rely on manual intervention for configuring and performing model transformations. [4] proposes the integration of model-driven development and aspect-oriented concepts. The introduced notion of *positive variability* refers to a core model to which selectively certain parts are added. The difference of this notion to Δ-designs/implementations is that the latter can also contain modifications and removals of design and implementation artifacts. Model transformations in [4] are realized by aspect-oriented composition of artifacts which also extends to the implementation by means of aspect-oriented programming concepts. However, the manual implementation of certain product parts is explicitly included in the approach which is not considered in our framework for automated product derivation.

Most approaches for automated product derivation consider only the design layer or the implementation layer. For automated derivation of product designs, [8] proposes an approach to automatically generate UML class and activity diagrams via annotations from variability models of the complete product line. In [9], product architectures are automatically derived from a common domain architecture model by means of model transformations. [16] considers an automated derivation of UML class diagrams by resolving explicitly specified feature-class dependencies.

There are different technologies for automated code generation applied in the context of software product lines, such as conditional compilation, frame technology [12], [17], generative programming [18] or code annotations [19]. Also, compositional approaches, such as aspect-oriented programming [20], feature-oriented programming [21] or mixins [22], are used to automatically generate product implementations from reusable artifacts. However, in order to generate products, it is assumed that the necessary code artifacts already exist. A systematic process how to design these artifacts is not provided.

The model-based framework for automated product derivation presented in [23] is structurally similar to the framework proposed in this paper. It contains a modeling layer describing the relation between product features and implementation artifacts. Because the implementation is based on aspect-oriented programming, the models define how classes and aspects are composed for feature con-

figurations. Product derivation is fully automated, but in contrast to the work presented in this paper, the approach is conceptually restricted to aspect-oriented techniques. This limits the means for dealing with product variability to the expressiveness of aspect-oriented concepts that can, for instance, not deal appropriately with features removing code.

III. Model-Based Automated Product Derivation

In order to provide a standardized technique how to design and implement product line artifacts suitable for automated product derivation, we propose a model-based framework. This approach is based on a model-based design layer that links product line variability declared in a feature model with the underlying implementation layer.

Overview. The proposed approach is structured into three layers (see Figure 1). During domain engineering, the variability of the software product line is captured by a feature model on the feature layer. Based on the feature model, reusable design and code artifacts are developed representing the product line variability on the underlying design and implementation layers. The design and the implementation artifacts are separated into a core design/implementation and Δ-designs/implementations, respectively, that can be configured automatically for a specific feature configuration during application engineering. The design concepts can be chosen such that relevant system aspects in each design stage can be adequately expressed. The design layer is independent of a concrete implementation technique, but provides the structure of the implementation artifacts. Designs can be refined based on the principles of model-driven development [10], until they are detailed enough for implementation. A product line design can be validated and verified before the development of code artifacts such that errors can be corrected less costly.

Feature Layer. The products of a software product line are described by a feature model. Features can represent functional behavior of products, but can also refer to non-functional aspects, such as deployment issues. A feature model declares the configurative variability of the product line, i.e., the commonalities of all products are captured by mandatory features, possible variabilities are modeled by optional features, and constraints between features are defined. The set of possible products of a product line is described by the set of valid feature configurations.

Design Layer. The design of a product line is split into a core design and a set of Δ-designs that are developed during domain engineering. The core design corresponds to a product of the product line with a basic set of features. This core can be developed according to well-established single application design principles. The variability of the product line is handled by Δ-designs. The Δ-designs declare

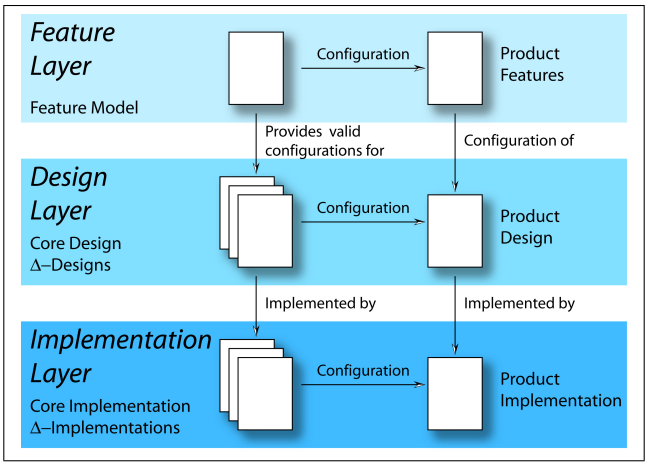

Figure 1. Model-based Automated Product Derivation

modifications to the core design in order to represent specific product characteristics. The step from the feature model to the design artifacts is a creative process because product line variability can be represented in different ways in a design.

In order to find a core design for a product line, a suitable basic feature configuration has to be identified. *Mandatory features* are always contained in the basic configuration, as they have to be present in all valid configurations. For *optional features*, the guideline adopted is that Δ-designs should add rather than remove functionality. If an optional feature only adds entities to the design, the feature should not be a part of the basic configuration. However, if an optional feature is included in many products, adding it to the core configuration can be beneficial because it can be tested thoroughly without considering product line variability. If selecting an optional feature causes that functionality is excluded from products, this feature should be contained in the core configuration to keep the core as small as possible. *Alternative features* represent options where at least one or exactly one feature has to be included in a valid configuration. Since the core configuration has to be valid, a choice between these options is necessary. If a feature selection requires to pick *at least* one feature, for the core *exactly* one feature should be chosen. The decision which option to include in the core can be based on an estimation which feature is most likely contained in many configurations.

Δ-designs define modifications of the core design to incorporate specific product characteristics. The modifications caused by Δ-designs comprise additions of design entities, removals of design entities and modifications of the existing design entities. The Δ-designs contain *application conditions* determining under which feature configurations the specified modifications have to be carried out. These application conditions are Boolean constraints over the features contained in the feature model and build the connection

between features in the feature model and the design level. A Δ-design does not necessarily refer to exactly one feature, but potentially to a combination of features. For example, if the feature model contains two features A and B, the constraint $(A \wedge \neg B)$ attached to a Δ-design denotes that the modifications are only carried out for a feature configuration if feature A is selected and feature B is not selected.

The general application constraints allow very flexible Δ-designs as combinations of features can be handled individually. The number of Δ-designs that are created for a feature model depends on the desired granularity of the application conditions. The application conditions of all Δ-designs can be checked if all features are addressed in at least one design. In order to obtain a design for a particular product during application engineering, all Δ-designs whose constraints are valid under the respective feature configuration are applied to the core. This can involve different Δ-designs that are applicable for the same feature in isolation as well as in combinations with other features. To avoid conflicts between modifications targeting the same design entities, first all additions, then all modifications and finally all removals are performed.

Implementation Layer. In order facilitate automated product derivation, the structure of the design is mapped to the structure of the implementation artifacts that are developed during domain engineering. The implementation artifacts are separated into a core implementation and Δ-implementations. The core design is implemented by the core implementation. As the core design is a complete product, single application engineering methods can be applied for implementing the core. This implementation can also be validated and verified thoroughly by well-established principles. Δ-designs are implemented by Δ-implementations which have the same structure as the Δ-designs. The additions, modifications and removals of code specified in Δ-implementations capture the corresponding additions, modifications, removals declared in the Δ-designs. The application condition attached to a Δ-implementation determines under which feature configurations the code modifications are to be carried out. The conditions directly refer to the application condition of the implemented Δ-designs. The process to obtain a product implementation for a specific feature configuration during application engineering is the same as for the design. The modifications specified by all Δ-implementations with a valid application conditions under a specific feature configuration are applied to the core. Again, first all additions, then all modifications and finally all removals of code are carried out. This analogous priority rule ensures that a product implementation generated for a specific feature configuration is an implementation of the corresponding product design.

The close correspondence between design layer and implementation layer provides a general approach to create reusable artifacts during domain engineering that suitable for automated product derivation during application engineering. The design layer provides the structure for the corresponding code artifacts. Because core design and core implementation are complete products, they can be developed by well-established principles from single application engineering. The independence of Δ-designs and Δ-implementations from core designs and core implementations, respectively, yields the potential of incremental, evolutionary product line development. Refinement of designs along the lines of model-driven development can easily be incorporated into the proposed framework, because refinement is orthogonal to the concepts for capturing product line variability. Since the design layer is independent of the implementation layer, the proposed model-based framework can be used with different concrete implementation techniques, as long as the concrete implementation technique allows expressing the desired modifications and supports automatic code generation.

IV. A Framework for Model-based Automated Product Derivation

In order to evaluate the proposed approach, we realized the model-based framework for automated product derivation for developing an information system product line. As application domain for the product line, we use the Common Component Modeling Example (CoCoME) [24] that describes a software system for cash desks dealing with payment transactions in supermarkets. Information systems involving clients-server communications are generally distributed and highly concurrent. To deal with this inherent complexity, we implement our system in the object-oriented, data-centric CoBox component and concurrency model [11]. A CoBox is a runtime component consisting of a (non-empty) set of runtime objects, i.e., other CoBoxes or instances of ordinary classes. Each CoBox at runtime executes a set of tasks, of which at most one can be active at any time. A task is active as long as it has not finished or willingly suspends its execution. Thus, inside a CoBox all code is executed sequentially. A CoBox communicates with other CoBoxes outside of its own CoBox via asynchronous messages. CoBoxes allow flexible deployment because the location where a CoBox is instantiated does not influence its functional behavior. This allows considering also variability of deployment besides variability of functionality in the product line to be developed

A. Feature Layer

For representing product line variability on the feature layer, we use feature diagrams [25]. In a feature diagram, the set of possible product configurations is determined by a hierarchical feature structure. A feature can either be mandatory, if it is connected to its parent feature with a filled circle, or optional, if it is connected with an empty circle. Additionally, a set of features can form an alternative

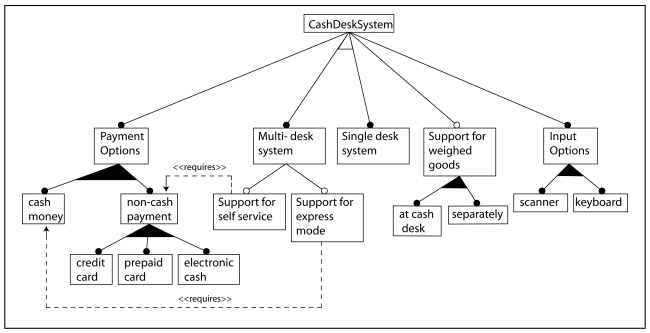

Figure 2. Feature Model for the CoCoME Software Product Line

selection in which at least one (filled triangle) or exactly one (empty triangle) feature has to be included in a valid configuration. Furthermore, constraints between features can be represented by explicit links.

To use CoCoME [24] as an example for a product line, we extended the application scenario with functional and deployment variabilities keeping the original system as one possible configuration. The feature model for the CoCoME software product line is shown in Figure 2. A CoCoME system has different payment options. First, it is possible to pay by cash or by one of the non-cash payment options, i.e., credit card, prepaid card or electronic cash. At least one payment option has to be chosen for a valid configuration. Product information can be input 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 themselves or at separate facilities. With respect to deployment, there is the alternative option to have a single-desk system with only one cashier or a multi-desk system with a set of cashiers. The multi-desk system can optionally comprise an express mode which requires cash payment or a self-service mode requiring non-cash payment.

B. Design Layer

Since we aim at a CoBox-based design and implementation of the product line, the design layer has to capture all relevant aspects for specifying CoBoxes. This includes the CoBoxes that classes belong to as well as deployment information for the CoBoxes. We introduce an extension to UML class diagrams [26] to express the additional information. Usually, UML diagrams are extended by stereotype annotations. This, however, would drastically impair the readability of the diagrams. With the extended notation, a CoBox design consists of a set of CoBoxes and ordinary classes. Graphically, CoBoxes are represented by a rounded box named the same as the owning CoBox class. Ordinary classes are denoted as usual UML classes. Both, CoBox classes and ordinary classes have member variables and methods. CoBoxes can contain other CoBoxes and other ordinary classes. UML relations describe relations between CoBoxes and classes. In addition, deployment information

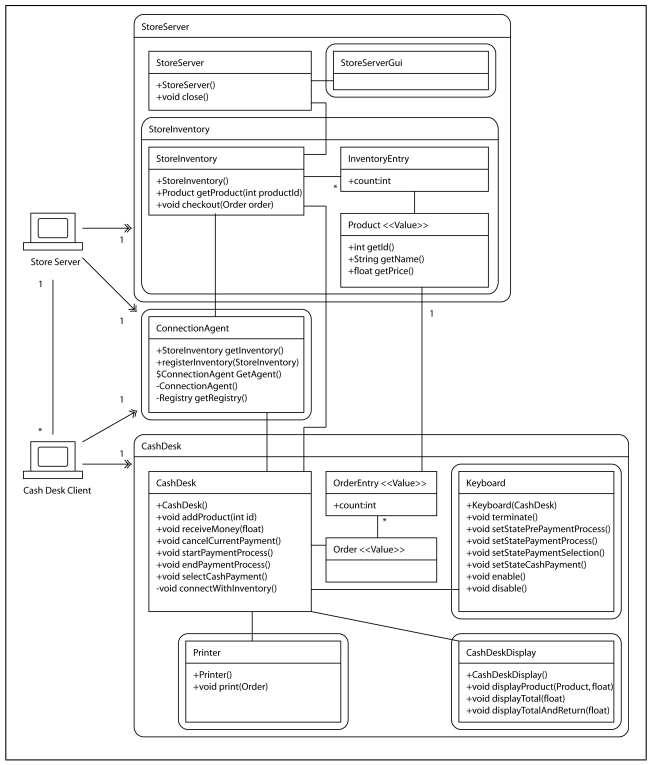

Figure 3. Core Design for the CoCoME Software Product Line

is provided by determining on which *deployment targets* CoBoxes should be instantiated. This is expressed by a doubled-headed arrow from a deployment target to a CoBox.

The design layer handles the variability of the feature model by a core design and a set of Δ-designs. The core design of a CoBox-based product line is denoted by a CoBox design. Δ-designs require additional notation to specify the modifications to the core design. In a Δ-design, it is defined which CoBoxes or classes are added or removed and which member variables or methods in existing CoBoxes or classes are added, removed or modified. The $+$ symbol marks additions, $-$ marks removals and $*$ denotes modifications. As UML class diagrams already use the $+$ and $-$ symbols for public and private members, we attach the alteration symbols to the right top corner of an altered CoBox, of an altered class or of an rectangle surrounding the altered class members. Additionally, each Δ-design contains its application condition, a Boolean constraint over the features in the feature model, to determine for which configurations the Δ-design is applied to the core. The application condition is displayed in an angular box at the top of the design.

The core configuration of the CoCoME software product line includes cash payment, keyboard input, and is a multi-desk system because cash payment and keyboard input are features of almost any cash desk system and most shops comprise more than one cashier. Other optional features are not incorporated into the core in order to keep it as

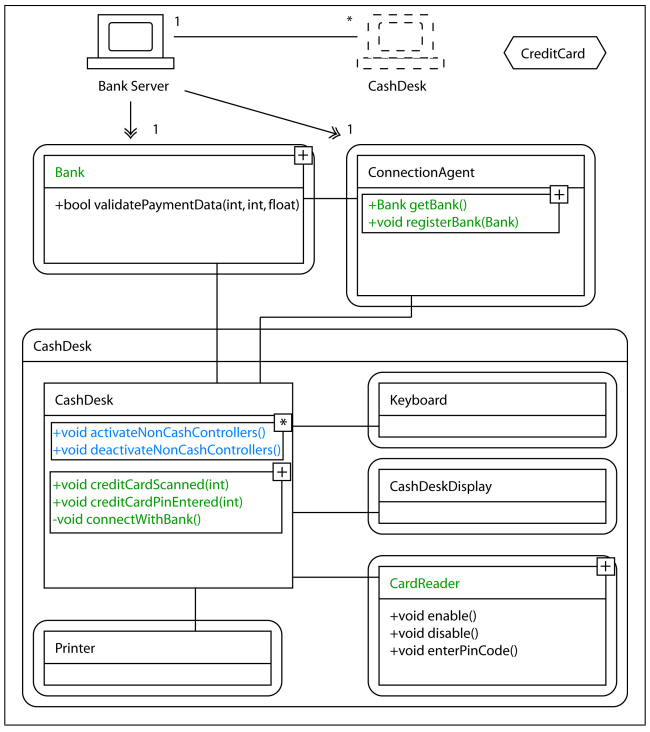

Figure 4. Δ-Design for Credit Card Payment

small as possible. The resulting CoBox core design is shown in Figure 3. The design specifies the `CashDesk` and `StoreServer` CoBoxes for realizing the core functionality. Instances of the `CashDesk` and `StoreServer` CoBoxes are created on the deployment targets `Cash Desk Client` and `Store Server`, respectively, that have to be physically connected. The logical connection is established via an additional `ConnectionAgent` CoBox created on each deployment target.

Figure 4 depicts the Δ-design containing the modifications for credit card payment, that is not included in the core configuration. To provide credit card functionality, a CoBox `Bank` has to be added to the system. Further, the CoBox `CashDesk` has to be extended by a CoBox `CardReader` and further class members to take care of the credit card payment. Also, the `ConnectionAgent` gets further member variables and methods to handle the communication with the `Bank`. This is denoted by the + symbol attached to the respective classes and members. Additionally, two methods of the `CashDesk` CoBox are modified, which is shown by the ∗ symbol. Deployment information for the `Bank` CoBox is provided relative to the overall system. The deployment target `Bank Server` on which the `Bank` CoBox is to be instantiated has to establish a physical connection to the deployment target on which `CashDesk` CoBox is executed. This allows dealing with deployment modifications caused by other Δ-designs. The angular box in the top right corner of the Δ-design shows

```
<x-frame name="CashDesk.CORE">
public cobox class CashDesk {
  ...
  private Keyboard _keyboard;
  private Order _currentOrder;
  <break name="CashDesk_AdditionalAttributes"/>
  ...
  public void selectCashPayment() {
    _keyboard!setStateCashPayment();
  }
  ...
  <break name="CashDesk_AdditionalMethods"/>
}
</x-frame>
```

Listing 5. Core Frame for the `CashDesk` CoBox

the application condition. This condition determines that the Δ-design is applied in all feature configurations in with the `CreditCard` feature is included. During application engineering, we obtain a design for a multi-desk system containing cash payment, credit card payment and keyboard input by applying the modifications specified in the Δ-design to the core design. This allows performing model-based validation and verification of this product already on the design level before the implementation is derived.

C. Implementation Layer

The implementation layer for the CoCoME software product line is realized by frame technology [12]. Frames structure source code into parts with pre-defined break points. The break points can be adapted by inserting code from other frames or by removing code from break points. In our model-based framework, the structure of the CoBox-based design is directly mapped to the frame structure on the implementation layer. The core design of a product line is realized by a set of core frames. Each CoBox in the core design is implemented by a core frame. The code in this core frame also contains break points that are necessary for modifications caused by Δ-frames. For each Δ-design in which the CoBox is altered, a Δ-frame is constructed that contains the respective modifications to this CoBox. Additionally, for each CoBox newly created by a Δ-design, a Δ-frame is generated that contains its implementation. The application conditions of the Δ-frames are the same as the ones of the implemented Δ-designs. Special build frames capture in which feature configurations the modifications of a Δ-frame are applied to the core frames.

XVCL [17] is a programming language-independent implementation of frame technology using an XML-dialect for defining frames, break points and break point adaptations. We use XVCL to realize the implementation layer of the CoCoME product line. The XVCL core frame for the `CashDesk` CoBox is depicted in Listing 5. This frame implements the design specified in Figure 3. The frame contains XVCL `break` tags for including additional attributes and methods that are specified by feature frames targeting this core frame. The Δ-frame in List-

```
...
<insert-after break="CashDesk_AdditionalMethods">
...
public void creditCardPinEntered(int pin) {
  _cardReader!disable().await();
  BankInterface bank = connectWithBank();
  if (bank == null) {
    System.out.println("CashDesk: Bank not available.");
    _cardReader!enable();
    return;
  }
  if (bank!validatePaymentData(_currentCreditCardNumber,
      pin, _currentOrder.price).await()) {
    receiveMoney(_currentOrder.price);
  } else {
    System.out.println("CashDesk: Unable to verify pin.");
    _cardReader!enable();
  }
}
</insert-after>
```

Listing 6. Δ-Frame for the `CashDesk` CoBox for the Credit Card Feature

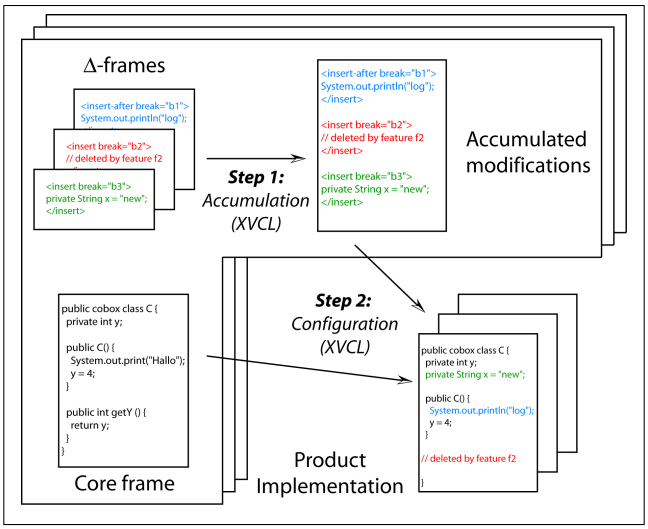

Figure 7. Automated Product Derivation using XVCL

ing 6 is an XVCL frame corresponding to the modifications applied to the `CashDesk` CoBox for the Credit Card feature that is specified in the Δ-design in Figure 4. Among other modifications, this Δ-frame defines that the `CashDesk_AdditionalMethods` break point in the `CashDesk` core frame has to be adapted by inserting the `creditCardPinEntered` method if the Credit Card feature is part of the configuration to be implemented.

The code for a product implementing a particular feature configuration can be automatically derived from the core frames and Δ-frames of the product line implementation during application engineering by the two-step derivation process depicted in Figure 7. Adapting core frames as it is necessary for automated product derivation is not possible in a single XVCL run. XVCL frames are defined in a tree-structured frame hierarchy. This structure is traversed (in-order) during processing, such that adaptations in superordinate frames overwrite adaptations in subordinate frames, if both frames target the same break point. This is useful for flexible specialized frame adaptations, but in our application, frames adapting the same break point should not modify each other. Therefore, in the two-step derivation process, first, the modifications specified in the Δ-frames with valid application condition are accumulated into a single temporary modification frame by one run of the XVCL processor. The selection of the Δ-frames contributing to the accumulated modifications is controlled by special build frames capturing the application conditions of the Δ-frames. In the second processing step, the accumulated modifications in the temporary modification frames are applied to adapt the corresponding core frames by a second run of the XVCL processor. This ensures a flat frame hierarchy for the second process, so that a set of modifications targeting the same break point are accumulated instead of being overwritten. The result of this process is a product implementation for the desired feature configuration.

V. Evaluation

We realized the CoCoME product line with the proposed model-based framework for automated product derivation [27]. The CoBox-based implementation of the product line is carried out in JCoBox[1] that is compiled to standard Java. As XVCL is programming language-independent, it is straight-forward to use it for the JCoBox implementation of the CoCoME product line. In the current implementation, 168 different products of the CoCoME product line can be derived automatically by the XVCL-based two-step derivation process. The implementation of the CoCoME software product line consists of 12 core frames, 21 Δ-frames and 12 build frames. The derivation process requires 6 additional meta frames not containing any source code to guide the derivation. Non-variable system parts are implemented in 5 regular source code files.

The advantage of the presented approach is that it is not limited to a particular implementation language or technique and applicable in a variety of scenarios. The development of the product line core allows using established single application engineering principles. Manual product-specific intervention is explicitly avoided such that modifications in any of the product line artifacts can be fully automatically propagated to existing products. There is no need for additional customization of products after product derivation. Modifications of the product line artifacts, however, affect all three layers. This introduces the need for additional synchronization mechanisms in case of parallel modifications.

VI. Conclusion

We have presented a model-based framework for automated product derivation relying on an independent model-based design layer. The design bridges the gap between

[1] http://softech.informatik.uni-kl.de/Homepage/JCoBox

feature models and product implementations. Its structure guides the development of implementation artifacts capable of automated product derivation. We realized and evaluated the proposed framework with an extended version of UML for the design and frame technology for the implementation.

For future work, we will realize the introduced framework with different implementation techniques to evaluate its general applicability. A first candidate is the trait-based language presented in [28]. Additionally, we will improve the tool support following our prototypical implementation. In order to analyze the effects of product line evolution for automated product derivation, we will formalize our approach to give a formal account how the design and implementation layers are affected by newly added features.

REFERENCES

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

[2] S. Deelstra, M. Sinnema, and J. Bosch, "Product Derivation in Software Product Families: A Case Study," *Journal of Systems and Software*, vol. 74, no. 2, pp. 173–194, 2005.

[3] C. W. Krueger, "New Methods in Software Product Line Development," in *SPLC*, 2006, pp. 95–102.

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

[5] J. Bayer *et al.*, "PuLSE: a Methodology to Develop Software Product Lines," in *Symposium on Software Reusability (SSR)*, 1999, pp. 122–131.

[6] C. Atkinson *et al.*, *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

[7] J. D. McGregor, "Preparing for Automated Derivation of Products in a Software Product Line," Carnegie Mellon Software Engineering Institute, Tech. Rep., 2005.

[8] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," in *Generative Programming and Component Engineering (GPCE)*, 2005, pp. 422 – 437.

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

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

[11] J. Schäfer and A. Poetzsch-Heffter, "CoBoxes: Unifying Active Objects and Structured Heaps," in *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2008)*, 2008, pp. 201–219.

[12] P. G. Bassett, *Framing Software Reuse: Lessons From the Real World*. Prentice Hall, 1996.

[13] 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.

[14] 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, pp. 109–114.

[15] 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, pp. 709–714.

[16] H. Gomaa and M. E. Shin, "Automated Software Product Line Engineering and Product Derivation," in *HICSS*, 2007.

[17] H. Zhang and S. Jarzabek, "XVCL: A Mechanism for Handling Variants in Software Product Lines," *Science of Computer Programming*, vol. 53, pp. 381–407, 2004.

[18] U. W. Eisenecker and K. Czarnecki, *Generative Programming*. Addison-Wesley, 2000.

[19] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," in *International Conference on Software Engineering (ICSE)*, 2008, pp. 311–320.

[20] G. Kiczales *et al.*, "Aspect-Oriented Programming," in *European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 1997.

[21] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," in *International Conference on Software Engineering (ICSE)*, 2003, pp. 187–197.

[22] Y. Smaragdakis and D. S. Batory, "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-based Designs," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 215–255, 2002.

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

[24] 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.

[25] 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.

[26] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

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

[28] L. Bettini, V. Bono, F. Damiani, and I. Schaefer, "Implementing Software Product Lines using Traits," Dipartimento di Informatica, Università di Torino, Tech. Rep., 2009, available at *http://www.di.unito.it/~damiani/papers/isplut.pdf*.