MAPLE2009

**Proceedings of the
1st International Workshop on
Model-Driven Approaches in
Software Product Line Engineering**

Goetz Botterweck, Iris Groher, Andreas Polzer,
Christa Schwanninger, Steffen Thiel,
Markus Voelter (Eds.)

August 24, 2009, San Francisco, California, USA

Abstract

Many of the benefits expected from software product lines are based on the assumption that the additional investment in setting up a product line pays off later when products created. However, to fully exploit this we need to optimize application engineering processes and handle SPL artifacts in a systematic and efficient manner. This workshop explores how model-driven approaches can help to achieve these goals. In particular the workshop revolves around three themes:

1. **Efficient product derivation.** The true return on investment in product line engineering, is achievable when the product lines can be efficiently used for product derivation. How can application engineering benefit from model-driven and aspect-oriented approaches?
2. **Link PLE research and industry practice.** We have to overcome the gap between research and industrial practice so that both sides can learn from each other. Hence, we are particularly interested in experience reports that discuss the use of models in *real world* PLE projects.
3. **SPL models with a meaning.** If we want to improve product derivation, we require models that are more than just vehicles for documentation and discussions on the whiteboard – models that are precise and expressive enough to be used in for automation and in advanced interactive tools. However, if the existing models are documentary and ambiguous, how do we come to more precise models?

Organisation

Editors

Goetz Botterweck	Lero, University of Limerick, Limerick Ireland
Iris Groher	Johannes Kepler University, Linz Austria
Andreas Polzer	RWTH Aachen, Germany
Christa Schwanninger	Siemens AG, Germany
Steffen Thiel	Furtwangen University, Germany
Markus Voelter	Independent consultant, itemis, Germany

Program Committee

Danilo Beuche	pure-systems, Germany
Goetz Botterweck	Lero, University of Limerick, Ireland
Paul Gruenbacher	JKU, Linz, Austria
Iris Groher	JKU, Linz, Austria
Holger Eichelberger	University of Hildesheim, Germany
Ulrich Eisenecker	University of Leipzig, Germany
Florian Heidenreich	TU Dresden, Germany
Bernhard Hollunder	Furtwangen University, Germany
Kyo-Chul Kang	POSTECH, Korea
Stefan Kowalewski	RWTH Aachen, Germany
Kwanwoo Lee	Hansung University, Seoul, South Korea
Johannes Mueller	University of Leipzig, Germany
Andreas Pleuss	Lero, University of Limerick, Ireland
Andreas Polzer	RWTH Aachen, Germany
Klaus Schmid	University of Hildesheim, Germany
Christa Schwanninger	Siemens AG, Germany
Steffen Thiel	Furtwangen University, Germany
Markus Voelter	Independent consultant, itemis, Germany
Jens Weiland	Reutlingen University, Germany

Contents

Tracking Evolution in Model-based Product Lines

Wolfgang Heider Rick Rabiser Deepak Dhungana Paul Grünbacher
Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University
Linz, Austria
{heider | rabiser | dhungana | gruenbacher}@ase.jku.at

***Abstract*— Software product lines are complex and need to be maintained and evolved over many years. New customer requirements, new products derived, technology changes, and internal enhancements lead to continuous changes of the artifacts and models constituting a product line. Managing such changes therefore becomes a key issue during a product line's evolution. We propose an approach that supports multi-level monitoring of product line artifacts and models and continuous tracking of changes. We present tool support for evolution tracking in Eclipse workspaces and illustrate our approach with examples from DOPLER, an existing Eclipse-based product line environment.**

Keywords-product line engineering; evolution; change tracking

I. Introduction

Product lines are typically used for many years and are inevitably subject to continuous evolution. Managing the evolution is success-critical for any product line approach as engineers need to deal with changes and extensions to the product line's assets and the derived products [1]. Feature models [2], decision models [3], extended UML [4], or aspect oriented approaches [5] are typically applied to define product lines. Managing the evolution of models therefore becomes a major concern.

In particular, our research interest is on (i) monitoring and tracking changes to models and product line artifacts, and (ii) establishing traceability between diverse product line artifacts such as product-specific requirements, change requests, or bug reports. Numerous research prototypes and commercial tools are available to support the creation and utilization of product line models, e.g., [6, 7]. However, they provide only limited support for dealing with product line evolution.

A generic approach for tracking the evolution of heterogeneous artifacts and models is still not available. For instance, existing approaches and tools lack support for managing the evolution of product line models at multiple levels of granularity and for managing interdependencies between different product line artifacts. This becomes particularly critical in a multi-team environment if several application engineering projects are conducted in parallel. This can mean that multiple products are derived concurrently from different releases of a product line.

In this paper we propose an approach for evolution tracking which is based on a generic meta-model. The approach is supported by our tool EvoKing. We demonstrate the capabilities of EvoKing using an example of its integration with the DOPLER product line approach and tools [8].

II. A Meta-model for Tracking Product Line Evolution

Many software tools support change tracking at the file or code level. For instance, version control systems and file system journaling mechanisms allow keeping track of changes to artifacts at the file level. Development environments make use of these tools to support change-tracking at the code level. However, tracking changes at this level is tedious. Supporting evolution requires change-tracking at a higher level of granularity and abstraction. It is also important to understand the dependencies between changes. Furthermore, change-tracking needs to cover various types of artifacts such as models, model elements, or structured documents.

From a bird's eye view, tracking evolution is about understanding the changes that are made to different artifacts of interest and establishing traceability between these artifacts based on dependencies between changes. The events and conditions that lead to a certain change are usually as interesting as the change itself. We have devised a generic meta-model for tracking evolution, which comprises artifacts, events, and relations (cf. Fig. 1).

An **artifact** is an element which needs to be monitored to track and manage its evolution. Examples of product line artifacts are meta-models, models, model elements, solution space elements (e.g., reusable code assets), or change requests (e.g., requirements captured during application engineering [9]). In a product line environment, these artifacts are typically managed in files or parts of files. The nature of the artifacts is domain-specific and cannot be generalized. Our evolution meta-model (the top layer in Fig. 1) thus does not specify concrete artifacts such as feature models, configuration files, or component descriptions. Instead we use a layered approach: the generic meta-model defines the basic elements that are then refined to specific domains and technologies using custom artifacts. Fig. 1 (middle and bottom layer) shows examples of artifacts at multiple levels of abstraction, i.e., in Eclipse-based tools and in product line engineering. Events and relations are created and resolved by implementing the defined custom artifacts (cf. Section 3).

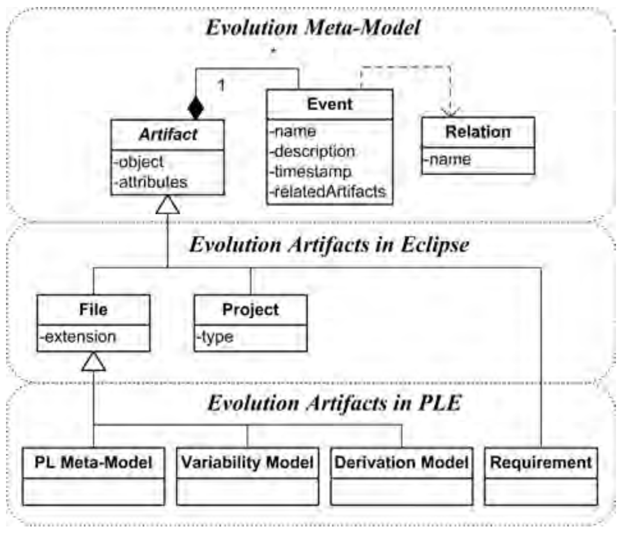


Figure 1. Evolution meta-model for tracking evolution and examples of custom artifacts for product line engineering artifacts in Eclipse.

An **event** causes one or more changes to artifacts. The generic evolution meta-model allows defining arbitrary events for the specified artifacts. Events relevant in product line engineering can typically be derived from existing product line process models and workflows. For example, the addition of a new variation point to a variability model constitutes an event that creates a new version of this model. Events can however also be defined at a much higher level of abstraction: e.g., if a user decides to derive a product using an existing variability model, a new application engineering project will be created, that is e.g., stored in a new model that needs to be tracked.

A **relation** between artifacts is established by an event tracked for specific artifacts. It describes how these artifacts are related with each other. Such links can be structural or temporal in nature. Structural relations between artifacts describe how the artifacts are organized, e.g., a model might be part of another model or a component might be described by a certain document. Temporal relationships are created at certain times during the artifact life-cycle to track their evolution history, e.g., a derivation model is created before a product is derived based on a variability model.

When refining our evolution meta-model to a particular product line development environment, users define different types of trace links as relations. Examples of relations (not shown in Fig. 1) between product line elements are:

- *Project to model*: A specific model (stored for example in a file) becomes part of a project and is marked for change tracking after its creation.
- *Model to model*: A model is related to another model. For instance, a variability model is based on a certain product line meta-model. Since multiple variability models and meta-models can be stored in a workspace it is necessary to establish traceability to ease product line evolution.
- *Model to model element*: A model consists of an arbitrary number of modeling elements.
- *Model element to model*: A model element can be related to different other models. For example, if a requirement is captured in a derivation model [10] or a requirements document during application engineering, it is useful to also establish a trace link from the requirement to the variability model that must be evolved to address the new requirement.
- *Model element to model element*: Model elements are typically related to other model elements. For instance, a newly captured requirement can directly refer to existing model elements like features, decisions, or assets in a product line model.

III. EvoKing: Tool-Support for Tracking Evolution in Eclipse Workspaces

Our approach for tracking and managing evolution of product lines is supported by our Eclipse-based tool EvoKing. We intentionally did *not* use Eclipse libraries to implement the evolution meta-model to keep the core of our approach independent from Eclipse. We describe the refinement of our generic evolution meta-model and the extensions we developed to support tracking of artifact changes in Eclipse.

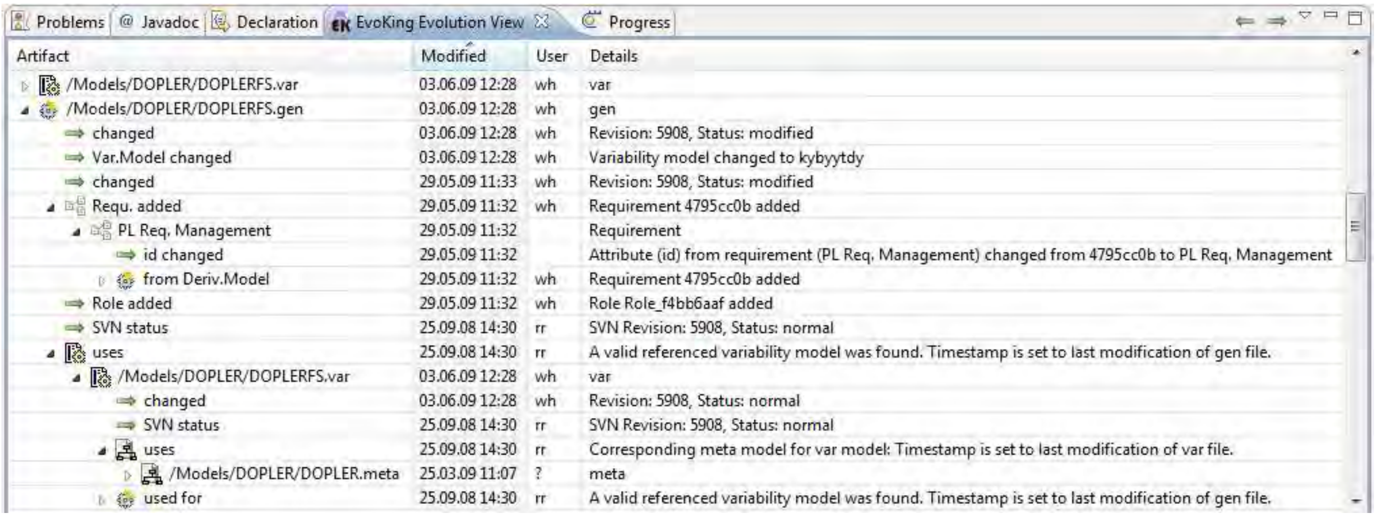
A. Refining the Meta-model for Eclipse

The **artifacts** tracked by EvoKing are Eclipse workspace entities like `IFile`, `IProject` or `IWorkbench`. They are defined in a refined evolution meta-model as shown in Fig. 1. Users configure EvoKing for an Eclipse-based modeling environment by specifying the artifacts of interest at a higher level of abstraction (the lower level implementation details like `IProject` or `IFile` are transparent to the user). For example, users specify the types of Eclipse projects they want to be tracked (e.g., “Java Project” or “Product Line Project”) or the file types (e.g., “Java source files” or “XY Models”).

Low-level **events** fired by the Eclipse framework (e.g., file change notifications) are automatically captured by EvoKing. EvoKing complements the existing notification mechanisms of Eclipse by adding an explicit meaning to events. For example, users can define in the evolution meta-model that whenever a new file of type “feature configuration” is added to the workspace, this shall be interpreted as the start of product derivation and a relation to a feature model should be created (see Section 4). This way a **relation** from a derivation project (i.e., stored in a feature configuration file) to a variability model (i.e., stored in a feature model file) is established.

B. Tool Architecture

EvoKing works as a consumer and recipient of event notifications coming from Eclipse or other custom event providers (cf. Fig. 3). Based on the incoming events and the



Artifact	Modified	User	Details
/Models/DOPLER/DOPLERFS.var	03.06.09 12:28	wh	var
/Models/DOPLER/DOPLERFS.gen	03.06.09 12:28	wh	gen
changed	03.06.09 12:28	wh	Revision: 5908, Status: modified
Var.Model changed	03.06.09 12:28	wh	Variability model changed to kybyytdy
changed	29.05.09 11:33	wh	Revision: 5908, Status: modified
Requ. added	29.05.09 11:32	wh	Requirement 4795cc0b added
PL Req. Management	29.05.09 11:32		Requirement
id changed	29.05.09 11:32		Attribute (id) from requirement (PL Req. Management) changed from 4795cc0b to PL Req. Management
from Deriv.Model	29.05.09 11:32	wh	Requirement 4795cc0b added
Role added	29.05.09 11:32	wh	Role Role_f4bb6aaf added
SVN status	25.09.08 14:30	rr	SVN Revision: 5908, Status: normal
uses	25.09.08 14:30	rr	A valid referenced variability model was found. Timestamp is set to last modification of gen file.
/Models/DOPLER/DOPLERFS.var	03.06.09 12:28	wh	var
changed	03.06.09 12:28	wh	Revision: 5908, Status: normal
SVN status	25.09.08 14:30	rr	SVN Revision: 5908, Status: normal
uses	25.09.08 14:30	rr	Corresponding meta model for var model: Timestamp is set to last modification of var file.
/Models/DOPLER/DOPLER.meta	25.03.09 11:07	?	meta
used for	25.09.08 14:30	rr	A valid referenced variability model was found. Timestamp is set to last modification of gen file.

Figure 2. EvoKing Evolution View showing the change history of a DOPLER derivation model (.gen file) and a related requirement, variability model (.var file) and meta-model (.meta file).

defined artifacts, new events with more detailed information regarding context and semantics can be generated. Such *evolution events* are then stored for each artifact and can be browsed using the EvoKing evolution view (cf. Fig. 2). Other tools implementing a specific interface can also be registered as an observer to retrieve evolution events if they wish to be informed about changes and their meaning.

EvoKing supports the user in further refining the evolution meta-model. This includes support for the modeler to add code for resolving relations, to interpret events from Eclipse for specific models, and to enrich change events with context-specific, semantic information. Product line engineers can thereby customize EvoKing to support evolution in arbitrary Eclipse-based product line environments.

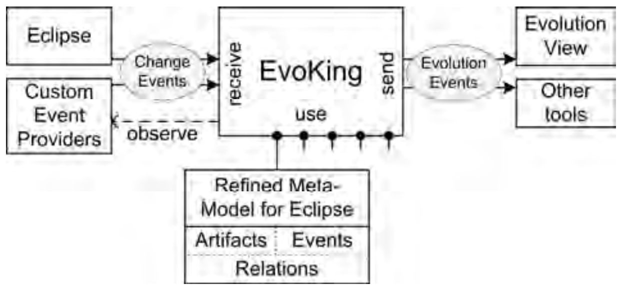


Figure 3. EvoKing's event architecture.

EvoKing recognizes change events based on information from two sources:

Eclipse resource change events such as `file added` or `file changed` and their sources are analyzed. EvoKing for example parses files representing models so that internal changes to models can be recognized using existing model APIs. Such changes are then mapped to artifacts and events defined in a refined evolution meta-model (see Section 4).

Custom event providers for models can send specific events to EvoKing. For example, if listeners have been implemented for a certain model type, they can be extended to explicitly fire change notifications. EvoKing is then registered as a listener for these models and can track changes being made to a model internally (e.g., model elements being added, deleted, or changed). Notifications are automatically transformed to `evolution events` according to the artifact and event definitions found in the evolution meta-model refined for a particular environment (cf. Section 4).

The EvoKing evolution view depicted in Fig. 2 shows all tracked artifacts of a project currently opened in Eclipse. The hierarchically organized representation of dependencies to other artifacts and all corresponding events allows users to quickly get an overview of the changes that have been occurring. Users can display details of a specific artifact at any time by expanding the tree, browsing through event details and related artifacts, and open editors for the elements the artifacts represent.

IV. Example Application of EvoKing: Evolution Management in DOPLER

Our testbed for EvoKing is the DOPLER product line engineering approach and tool suite [8]. We have been developing DOPLER in ongoing research collaboration with industry. The model-based, decision-oriented approach supports variability modeling and product derivation and provides tool support for creating, using, and managing diverse types of product line artifacts and models.

The product line artifacts (cf. Fig. 4) in DOPLER are product line meta-models, variability models, derivation models, and diverse model elements (e.g., assets, decisions, and product-specific requirements). The relevant dependencies between these artifacts are as follows: A variability model (`.var` file in Eclipse) uses a particular meta-model (`.meta` file); a derivation model (`.gen` file) is based on a specific variability

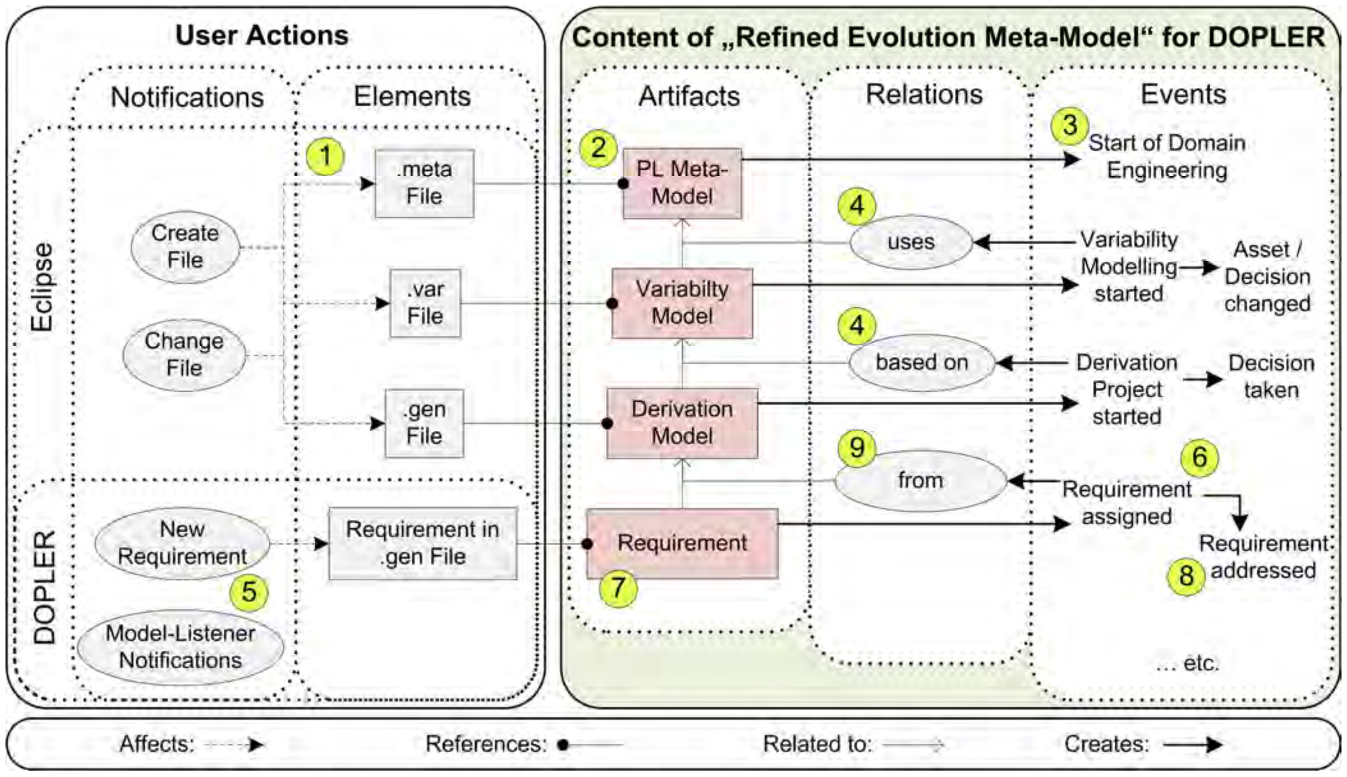


Figure 4. EvoKing customized for DOPLER. The left side shows elements and notifications we see within the workspace and editors. The right side shows artifacts, relations and events that represent the left side enriched with information taken from the refined evolution meta-model for DOPLER.

model; a requirement comes from a particular derivation model.

Evolution in DOPLER is for instance triggered by product-specific requirements captured during application requirements engineering. Requirements are captured in the derivation model representing a particular product derivation project. Implementing a requirement typically causes a change of the variability model (and thereby its elements like, i.e., assets and decisions).

Fig. 4 shows a simplified overview of how we customized EvoKing for DOPLER. Operations on files defined as model containers (`.meta`, `.var`, and `.gen` files) are captured and processed in the corresponding artifact implementations. For instance, for the creation of a `.meta` file (1) the artifact for the contained product line meta-model (2) is created. This leads to an evolution event indicating the start of domain engineering (3). This procedure works similar for other files and models. Starting variability modeling or starting a new derivation project additionally creates trace links between (4) the product line meta-model or variability model respectively. Independent of file changes, DOPLER-specific notifications are processed by the EvoKing artifacts. For instance, the DOPLER tool suite notifies EvoKing about model changes (5) like new model elements (i.e., assets, decisions, requirements) being added. EvoKing stores events containing this information (6) or, according to the refined evolution meta-model, new artifacts, (7) e.g., representing requirements, are held with their own evolution history (8) and relations to their origin (9).

EvoKing allows users to track the evolution of DOPLER product line meta-models, variability models, derivation models, and of the elements these models comprise. The customization of EvoKing to a different (Eclipse-based) product line environment would be pretty straightforward as most Eclipse-based product line environments store models in files in Eclipse projects and different model elements such as features or requirements are contained in the models.

V. Conclusions and Future Work

We presented a tool-supported approach for multi-level monitoring and tracking of changes to facilitate evolution in model-based product line engineering. Based on a generic meta-model for tracking evolution our tool EvoKing supports evolution management in Eclipse-based product line environments. We illustrated the applicability of our approach by customizing EvoKing for the DOPLER product line tool suite.

EvoKing automatically maintains a development history showing what and when was done by whom during development. There are, however, more advanced usage scenarios for the tool which we plan to explore in the future. For instance, we will use of the refined evolution meta-model and evolution information tracked by EvoKing to assist users with their *workflow* of modeling and creating product line artifacts. We will also use the relations captured by EvoKing as trace links for the purpose of *consistency checking* in and between product line models and artifacts. This will help to point out potential update leaks or inconsistencies after changes to specified artifacts. We plan to improve support for *further development of artifacts and relations*. This way, for example, changes to configuration files, custom service configurations, and component interface definition files can be tracked to ease maintenance tasks. Finally, the information collected by EvoKing allows deriving product and process metrics to facilitate benchmarking, to monitor development processes, and to track variability shifts in product lines.

Acknowledgment

This work has been conducted in cooperation with Siemens VAI Metals Technologies and has been supported by the Christian Doppler Forschungsgesellschaft, Austria.

References

[1] D. Dhungana, T. Neumayer, P. Grünbacher, and R. Rabiser, "Supporting Evolution in Model-based Product Line Engineering, "Proc. of the *12th International Software Product Line Conference (SPLC 2008)*, Limerick, Ireland, IEEE Computer Society, 2008, pp. 319-328.

[2] K. Czarnecki and C. H. P. Kim, "Cardinality-Based Feature Modeling and Constraints: A Progress Report, "Proc. of the *International Workshop on Software Factories at OOPSLA'05*, San Diego, USA, ACM Press, 2005, pp. 1-9.

[3] K. Schmid and I. John, "A Customizable Approach to Full-Life Cycle Variability Management," *Journal of the Science of Computer Programming, Special Issue on Variability Management*, vol. 53(3), pp. 259-284, 2004.

[4] H. Gomaa, *Designing Software Product Lines with UML*: Addison-Wesley, 2005.

[5] M. Voelter and I. Groher, "Product Line Implementation using Aspect-Oriented and Model-Driven Software Development, "Proc. of the *11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, IEEE CS, 2007, pp. 233-242.

[6] A. Pasetti and O. Rohlik, "Technical Note on a Concept for the xFeature Tool," P&P Software GmbH / ETH Zurich, PP-TN-XFT-0001 2005.

[7] C. Krueger, "BigLever software gears and the 3-tiered SPL methodology, "Proc. of the *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'07)*, Montreal, Quebec, Canada, ACM, 2007, pp. 844-845.

[8] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer, "Integrated tool support for software product line engineering, "Proc. of the *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, Atlanta, Georgia, USA, ACM, 2007, pp. 533-534.

[9] R. Rabiser and D. Dhungana, "Integrated Support for Product Configuration and Requirements Engineering in Product Derivation, "Proc. of the *33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'07)*, Lübeck, Germany, IEEE Computer Society, 2007, pp. 219-228.

[10] R. Rabiser, P. Grünbacher, and D. Dhungana, "Supporting Product Derivation by Adapting and Augmenting Variability Models, "Proc. of the *11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, IEEE Computer Society, 2007, pp. 141-150.

Enabling End-users Participation in an MDD-SPL Approach

Francisca Pérez, Pedro Valderas, Joan Fons
Research Centre on Software Production Methods
Technical University of Valencia
Valencia, Spain
{mperez, pvalderas, jfons}@pros.upv.es

***Abstract*—Developing smart home systems that properly fit end-user needs is not always an easy task due to the lack of understanding that may exist between end-users and system developers. In the context of Software Product Lines, several approaches have been presented to improve the development of smart home system functionality. However, little support is provided to improve the interaction with end-users. In this work, we extend a Software Product Line based on Model-Driven Development with an interactive design tool that allows end-users to actively participate in the SPL. This tool allows end-users to configure the decision model that drives the production process of the software product line by themselves. In order to develop this tool we have been inspired by well-known and tested end-user techniques and interaction patterns that improve the user interface usability.**

I. Introduction

Smart home systems are in charge of providing different services to support the daily activities of the inhabitants of a home. In order to do this, smart home systems automatically perform actions such as turning the lights on [1], controllling a thermostat, closing the blinds, etc. However, all these actions must be performed according to the user's preferences and needs.

Adapting smart home systems to end-users needs is not always an easy task due to the lack of understanding that may exist between end-users and system developers. End-users are the owners of the domain of knowledge, the ones with more in-depth knowledge about both the services that must be provided by the system and the environment in which the system is going to be deployed. However, many times they do not have the ability of transmitting this information properly. We think that this can be improved by providing mechanisms that allow end-users to actively participate in the development process.

In the area of Software Product Lines (SPL), many efforts have already been made to improve the development of smart home systems [2], [3]. However, these approaches focus mainly on providing developers with techniques and tools to develop the system functionality, and they pay little attention to the interaction with end-users. In this work, we face the problem of allowing end-users to actively participate in the development of a smart home within an SPL.

To do this, we have extended a Software Product Line to develop smart home systems [4], which is based on Model Driven Development (MDD). The proposed extension consists of an interactive design tool that allows end-users to create tailored solutions that directly reflect their needs and expectations. To do this, we have been inspired by well-known and tested end-user techniques and interaction patterns that improve the user interface usability [5], [6], [7].

Considering the schema of the MDD-SPL (see Fig. 1), where a product operation transforms input assets into an output system according to the configuration specified in a decision model, the contribution of this work is an end-user tool that enables end-users to configure the decision model that drives the production process by themselves.

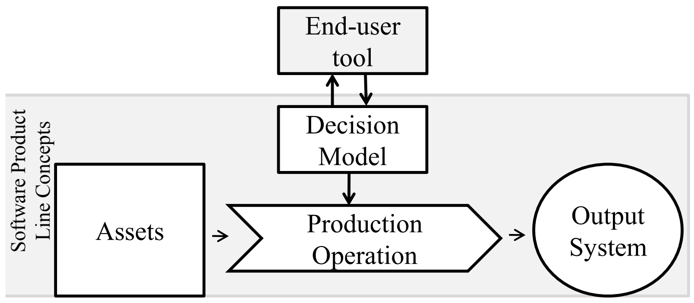


Fig. 1. Approach overview

The rest of this paper is structured in the following way: Section II presents the related work in the field of the end-user development techniques for smart homes. Section III presents the MDD-SPL for developing smart home systems. Section IV introduces the end-user tool and the interaction patterns that have been applied to improve the interface usability. Section V presents some aspects of the technology used to implement this tool. Finally, section VI concludes the paper.

II. Related work

There are several works that show how to combine MDD and SPLs [8], [2]. Voelter and Groher [2] describe an approach where development is combined with model-driven development. They define aspects at the modelling level, the transformation level, and the implementation level. They apply their approach to the Smart Home domain. Anastasopoulos et al. [8] apply a combination of both MDD and SPL to the Ambient Assisted Living (AAL) domain. They express variations in smart home functionality as features, and synthesize AAL specifications by composing features. Compared to our work, the above approaches do not involve end-user expectations in the MDD-SPL, which is essential for the successful development of Smart Homes [9]. Other works such as [10] presents a tool to support end-users in working with large-scale product line variability models in product derivation. This tool is based on derivation models and it provides end-users with a textual visualization which allows end-users to set values on decisions by answering questions. However, the use of a visual language seems to be the best option since visual languages have demonstrated to be more intuitive and easier to be used by users than other options like textual languages [11], [12].

Many research initiatives seek to allow end-users to program or customize their systems using end-user techniques as Pervasive Interactive Programming (PiP) [13], or CAPpella [14]. Furthermore, other research initiatives allow end-users to interact with their system using metaphors as jigsaw puzzle pieces [15], or magnetic refrigerator poetry [16]. Some of these well-accepted end-user techniques are:

- **Natural Programming [17]:** it is an application of the standard user-centered design process to the specific domain of programming languages and environments. The premise of this approach is that programmers will have an easier job if their programming tasks are made more natural. For example, HANDS [18] is a programming system for children. HANDS is an event-based system featuring a concrete model for computation based on concepts that are familiar with non-programmers. The computation is represented as an agent named Handy, sitting at a table handling a set of cards.
- **Programming By Example [19]:** also called Programming by demonstration (PBD) because the user shows examples of the desired behaviour to the computer. For example, Pervasive Interactive Programming (PiP) [13] provides a platform that uses the physical user space as the programming environment providing the user with a natural and more familiar mechanism to "program" the functionality they require to suit their particular needs.
- **Visual Programming [20]:** it is the use of visual expressions in the programming process. For example, Alice [21] is an innovative 3D programming environment that allows students to learn fundamental programming concepts in the context of creating animated movies and simple video games.
- **Jigsaw metaphor [15]:** it is based on the familiarity evoked by the notion and the intuitive suggestion of assembly by connecting pieces together. Essentially, it allows end-users to make variability decisions through a series of left-to-right couplings of pieces. For example, ACCORD has developed the Tangible Toolbox [22], based on a shared Data Space, that enables people to easily administer and re-configure services based on embedded devices around the home. This toolkit also enables devices to integrate with each other through several different editors. One of these editors uses the jigsaw metaphor to create new services.

Although these techniques encourage end-users to participate in the creation of software systems, they do not address a process where end-users can specify the requirements of the system. Our approach applies end-user techniques within an MDD-SPL in order to allow end-users to actively participate in the configuration of the desired software (in this particular case, a smart home system).

III. MDD-SPL for Smart Homes

In this section, we illustrate the SPL for smart home systems. Fig. 2 illustrates the models used in the SPL. The input assets consist of a collection of models describing all smart homes that can be produced. These models are created by using the PervML language. A smart home is uniquely defined by the selections made on the feature model, which plays the role of decision model. The selected features determine which elements of the PervML models are used for the initial configuration of the smart home by means of a Realization Model. Finally, the output system is obtained through a model transformation.

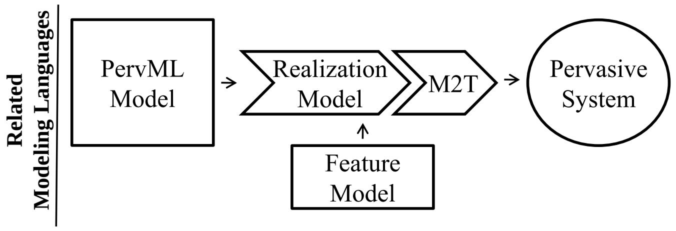


Fig. 2. MDD-SPL for Smart Homes

The following subsections provide details about the models involved in the SPL.

A. The PervML model

Pervasive Modeling Language (PevML) [23] is a DSL for describing pervasive systems using high-level abstraction concepts. This language focuses on specifying heterogeneous services in specific physical environments such as the services of a smart home. These services can be combined to offer more complex functionality by means of interactions. These services can also start the interaction as a reaction to changes in the environment. The main concepts of PervML are: (1) a *Service* coordinates the interaction between suppliers to accomplish specific tasks (these suppliers can be hardware o software systems); (2) a *Binding provider* (BP) is a supplier adapter that embeds the issues of dealing with heterogeneous technologies; (3) an *Interaction* is a description of a set of ordered invocations between Services; and (4) a *Trigger* is an ECA rule (Event Condition Action) that describes how a Service reacts to changes in its environment. This DSL has been applied to develop solutions in the smart home domain [24].

This model (see the bottom of Figure 3) describes the building blocks for the assembly of a pervasive system [23].

The grey blocks implement the functionality of the selected features. The white blocks enable an alternative functionality of the system. The (l), (o), (m) and (p) blocks provide adapters for the new resources available.

B. The feature model

Feature models are widely used to describe the set of products in a software product line in terms of features. In these models, features are hierarchically linked in a tree-like structure and are optionally connected by cross-tree constraints. There are many proposals for the type of the relationships and the graphical representation of feature models [25]. We have chosen the Feature Model [26] as the modeling language because it is feature reasoning oriented and has a good tool support [27].

This model (see the top of Fig. 3) determines the initial and the potential features of the smart home. The grey features are selected to specify a member of the smart home family. The white features represent potential variants. Initially, the smart home provides *Automated illumination, Presence simulation and a Security* system. This security system relies on *In home* detection (inside the home) and a siren alarm. The system can potentially be upgraded with volumetric presence detection and more alarms to enhance home security.

The feature model also determines how the features relate to each other by cross-tree constraints. As the feature model of Fig. 3 shows, these relationships are: **Optional** represented with a small white circle on top of the feature, **Mandatory** represented with a small black circle on top of the feature, **Multiple choice** represented with a black triangle, **Single choice** represented with a white triangle, **Requires** which it is represented with a dashed arrow and **Excludes** represented with a dashed double-headed arrow.

C. Realization model

The realization model is an extension that we have incorporated to Atlas Model Weaving (AMW) [28] in order to relate the SPL features to the PervML elements. AMW is a model for establishing relationships between models. Our extension augments the *AMW relationship* with the *default* and *alternative* tags. This augmented relationship is applied between features and PervML elements (BPs and Services). In the context of a BP, the *default* relationship means that the BP is selected for the initial configuration of the system. The *alternative* relationship means that the BP is considered a quiescent element that should be incorporated to the SPL product, but does not participate in the initial configuration. Quiescent BPs provide an alternative BP to replace the default BP in case of fault. The more quiescent BPs identified, the more flexible the adaptation will be.

This model (see the middle of Figure 3) establishes the relationships between the features and the PervML elements. For instance, the visual alarm feature is related to a BP (p) for visual alarms, but, alternatively, it can be replaced with a BP (m) that emulates the visual alarm by using the blink lighting.

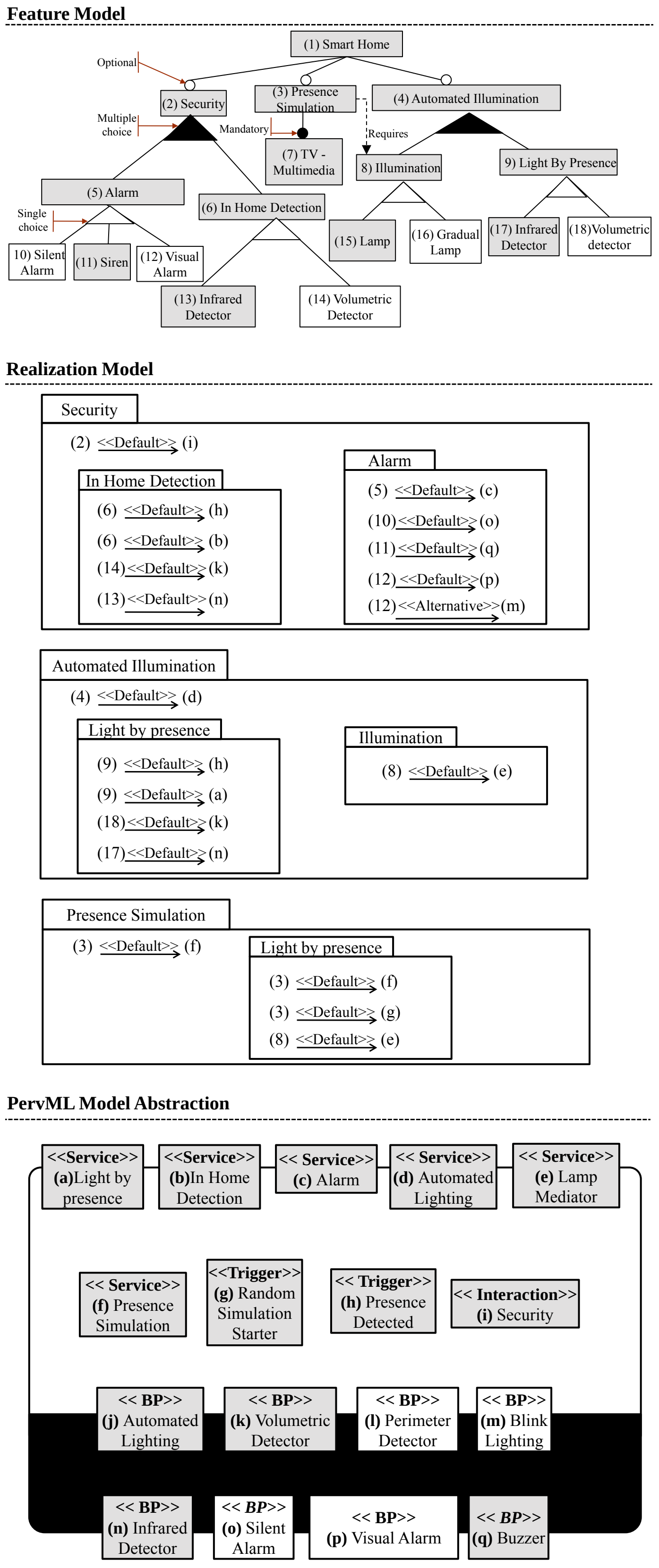


Fig. 3. Models for the SPL

D. Model To Text (M2T)

Once the pervasive system is modelled, the transformation engine can be applied to generate the code. For this task, we have used the MOFScript language which provides capabilities navigating models, creating files, etc. MOFScript takes as input one model and applies over one selected metaelement a contextual rule. The applied rule can access the element properties, navigate over the related model elements and invoke other rules.

At [1] there is more information about the transformation rules and the tools to support the code generation.

IV. Introducing end-users in the MDD-SPL

In the presented MDD-SPL, variability engineers set the smart home configuration by means of the feature model. Variability engineers make assumptions about the desirable functionality of end-users. Conversely, end-users are the ones who best know their activities and their functionality expectations. End-users and professional developers actually possess distinct types of knowledge. End-users are the "owners" of the problem and developers are the "owners" of the technology to solve the problem. End-users do not understand software developers' jargon and developers often do not understand end-users' jargon [29]. Although, end-users are not professional developers they have deep knowledge of their specific environment and they should be able to develop their own smart home system according to their needs. Hence, we involve end-users in the Smart Home configuration in order to minimize the mismatch between user expectations and system behaviour.

In order to tackle this, end-users must be supplied with visual development tools that allow them to describe their needs [30]. In this work, we have developed a tool that allows end-users to configure their smart home system using the MDD-SPL for smart homes described in the previous section. Fig. 4 shows an overview of the MDD-SPL with end-users. The end-user tool allows end-users to indicate which services and devices must be available in each location and configuring the feature model accordingly. Thus, when end-users have finished describing their needs, we obtain the decision model that determines the output system to be obtained by applying the model transformation.

To design the end-user front-end, we have based on well-accepted techniques and metaphors in the field of end-user development such as: Natural Programming, Programming By Example, Visual Programming and metaphors (see Section II). We have also applied interaction patterns and design principles to end-user interface design according to studies [5], [6], [7] which show how these patterns and principles help end-users (who may not have any background about computer applications). According to these studies, the main design interface decisions that we have applied are:

- **Using a wizard:** in our process the end-user needs to achieve a single goal (the description of their needed system) but several decisions need to be made before the goal can be fully achieved (several steps), which may not be known to the user. Thus, the use of a wizard is recommended in [5] since the user wants to reach the overall goal but may not be familiar with or interested in the steps that need to be performed.
- **Offering navigation buttons:** we use navigation buttons to suggest end-users that they are navigating a path with steps. This is recommended in [5] because the learning and memorization of the task of each step are improved. In addition, when users are forced to follow the order of tasks, they are less likely to miss important things and therefore will make fewer errors.
- **Displaying the elements using a grid layout:** this is recommended in [5] to any circumstance where several information objects are presented and arranged spatially within a limited area. This improves the presentation and it minimizes the time to scan, read and view objects on screen.
- **Offering options:** an interesting conclusion is reached in [6]: *what people see is what they select from!*. The study states that people tend to select from the entire list of options what they are first presented with. Rarely is an effort made to find additional options through scrolling. If eleven items are presented, the choice is from these eleven. When options must be compared among themselves, controls presenting all the options together will yield the best results.
- **Selection rather than introduce text:** the studies presented in [7] show the advantages and disadvantages of using either entry fields or selection fields for data collection. Since information became less familiar or subject to spelling or typing errors they recommend choosing a selection technique.

[1]www.pros.upv.es/labs/projects/pervml

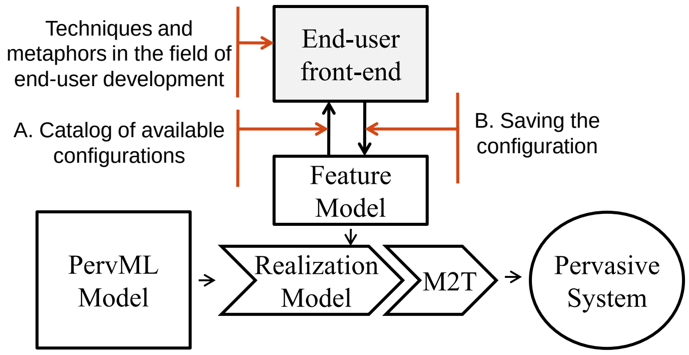

Techniques and metaphors in the field of end-user development
End-user front-end
A. Catalog of available configurations
B. Saving the configuration
Feature Model
PervML Model
Realization Model
M2T
Pervasive System

Fig. 4. Approach overview

Thus, we have developed a user interface based on the interface decisions presented above which allows end-users to specify the services and devices that they need. Fig. 5 shows a snapshot of this interface as end-users configure devices and services in their home. Each interface is divided into four areas: (1) Title and navigation buttons, (2) Catalog of available configurations, (3) End-user environment and (4) Information

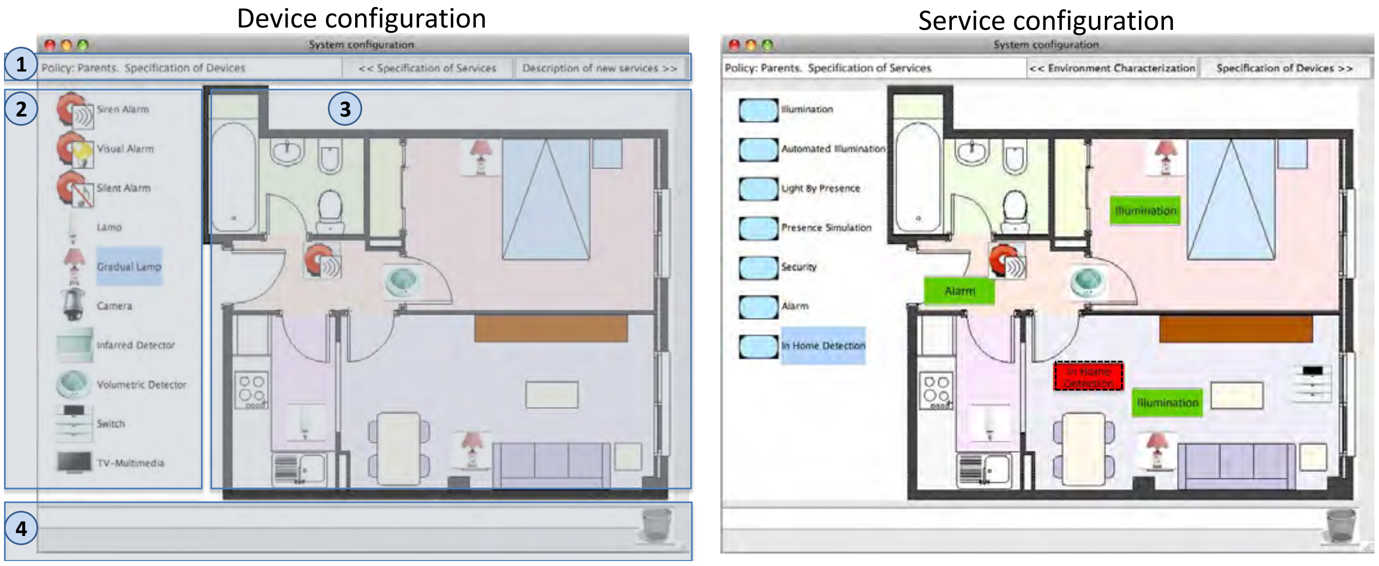


Fig. 5. Snapshot of the end-user front-end

where our tool can advise to end-users or assist them. In particular, we show at the left side of the figure how the end-user has selected some devices for different locations of their smart home (i.e. a siren alarm and a Volumetric detector for the corridor). At the right of the figure we show how the end-user has selected some services for different locations (i.e. Alarm service for the corridor).

As Fig. 5 shows, we have applied the interaction patterns described above. The *grid layout* pattern is applied to divide the interface into the four areas presented above. The *wizard pattern* is used to guide end-users along the process of creating a pervasive description by progressively asking them for the required information (services, devices, etc.). In addition, *navigation buttons* are also used in the area (1) to allow end-users to navigate between the different windows that ask for the required information. The *offer options* and *selection rather than introduce text* patterns are applied in the area (2) offering the devices/services available as options and allowing end-users to select these devices/services into the end-user environment represented in the area (3).

The next two subsections describe how the tool uses the Feature Model. Subsection A. describes how the end-user front-end uses the feature model to show the catalog of available configurations (see Fig. 4) and Subsection B. describes how the tool saves the configuration in the feature model activating/inactivating features according to the end-user's configurations.

A. Catalog of available configurations

As we described in subsection III-B, we use the feature model to describe the system configuration and its variants in terms of features. In the smart home domain, the system configuration that end-users have to select is made up of the services and devices required for each location in the environment.

At the top of Fig. 3 is shown the feature model which determines the initial and potential features of the smart home. These features represent services and families of devices. The **families of devices** are the leaves of the feature model and the **services** are the nodes which are not leaves. We specify families of devices in the feature model rather than devices because there is a large diversity of devices which are continuously changing. For each family of devices we offer a catalog of compatible devices. For example, the *Volumetric Detection* device family has a catalog of compatible devices which contains a Volumetric 360 degree detector as well as a 160 degree one. Thus, when a new device is supported all we have to do is update the catalog of that family of devices rather than the feature model.

Our tool shows end-users the options from the available configurations according to the feature model. Fig. 6 shows an example of service and device options according to the feature model. Note that these device options match the node leaves of the feature model presented in the figure (Siren Alarm, Visual Alarm, Siren Alarm, Infrared Detector and Volumetric Detector) and the service options match the nodes which are not leaves of the feature model (Security, Alarm and In Home Detection).

The available options are displayed in a tree. Studies described in [5] recommend using a tree when the number of groups is high. They also recommend that each option be explained so that users know of the consequences. Thus, we show in our tool a representative image for each service or device and a brief description. Fig. 5 shows the list of available devices and services that is shown to end-users from the feature model presented at the top of Fig. 3.

B. Saving the configuration in the feature model

Once the catalog of configurations has been shown, end-users can select services or devices for each location in

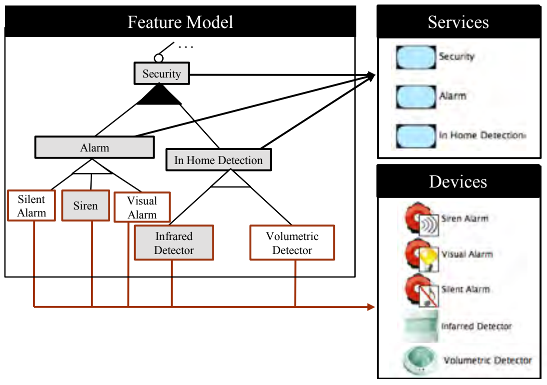


Fig. 6. The catalog of available configurations

the end-user's tool. When this happens, the end-user tool sets activated/inactivated features to the feature model. Thus, when end-users finish setting their system the feature model will have activated the features according to the end-user configuration.

To set a configuration end-users have to select the desired services and devices from the catalog of configurations and put them into the proper location. Then, a representative image of the service/device is displayed in the environment. The service/device can be displayed in two different colours: (1) red with a dotted frame if the service configuration has not fulfilled their constraints (services/devices that the service need) or (2) green if the service configuration has fulfilled their constraints. Fig.5 shows the specification of devices (see at the left side of the figure) and services (see at the right side of the figure).

In order to define these end-user interfaces we have based on the following end-user principles and interaction patterns [7], [5]:

- **Using autocompletion:** The study showed in [7] states that aided entry, also known as autocompletion, is preferred over unaided entry methods, and it is also the fastest method. Autocompletion reduces errors in comparison to unaided entry. In addition, it also minimizes the user's effort by reducing input time and keystrokes.
- **Using a warning:** this is recommended in situations where the user performs an action that may unintentionally lead to a problem [5] and the system cannot or should not automatically resolve this situation so the user needs to be consulted. The warning might also include a more detailed description of the situation to help the user make the appropriate decision by means of two options at least.
- **Offering all options:** this is recommended when the number of options is not large and they can be displayed without scrolling [7]. Rarely was an effort made to find additional options through scrolling.
- **Offering some options:** this is recommended when the number of options is high and it needs a scroll to be displayed. Thus, it is recommended to show some options of the available list [7]. This improves the speed of performance and satisfaction

According to the interaction patterns presented above, we have defined a set of mappings between the feature model and our end-user front-end and how the interaction patterns are used depending on the information that is available at the feature model. Next we present the interaction patterns used for each relationship of the feature model:

- If there is a **Mandatory feature**, we use **Autocompletion**. When a feature A is related to another feature B with a mandatory relationship, if A is selected B has to be selected too. In the end-user front-end, features are represented by services/devices. Thus, when the end-user selects a service that represents a feature A with a mandatory relationship to a feature B, the service representing feature B is automatically added to the same location of service A. For instance (see Fig. 7) when the end-user selects the *Presence Simulation* service for the living room (1) the *TV-Multimedia* device is automatically added to the same location (2) because there is mandatory relationship between the *Presence Simulation feature and the TV-Multimedia* feature. In addition, the feature model is updated by activating both features (3).

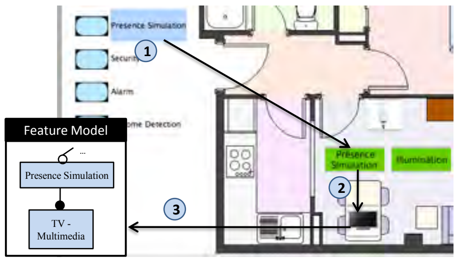


Fig. 7. Applying patterns in a mandatory relationship

- If there is a **Requires or Excludes feature**, we use **Warning**. When a feature A has a requires relationship with B, if A is selected feature B has to be selected too. Similarly, if feature A has an excludes relationship with B, when feature A is selected feature B does not have to be selected. In the end-user front-end, when the end-user selects a service that represents a feature with a requires or excludes relationship, the end-user front-end warns end-users by showing a warning. Fig. 8 shows when the end-user selects the *Presence Simulation* service for the living room (1). As the feature that represents this service has a requires relationship with the *Illumination* feature, the end-user front-end shows a Warning (2). Then the end-user adds this required service to the same location (3) and the feature model is updated activating the features *Illumination and Presence Simulation* (4).
- If there is an **Optional or single choice feature**, we use **Show all options and Autocompletion**. When a feature A has an *optional o single choice* relationship with other features, one of them has to be selected. In the end-user front-end, when the end-user selects a service

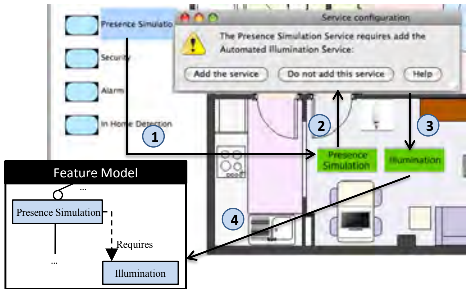
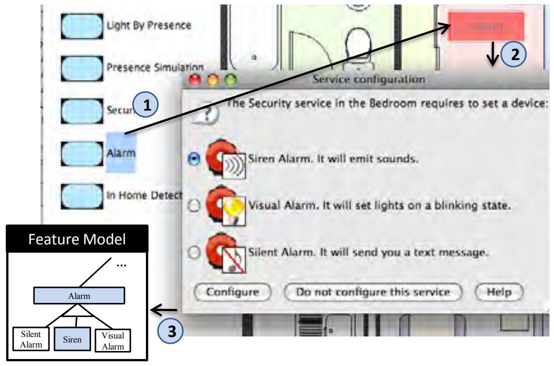

The Presence Simulation Service requires add the Automated Illumination Service:

Fig. 8. Applying patterns in a requires relationship

that represents a feature with an optional or single choice relationship, the end-user front-end shows a dialog with all the services/devices that represent the related features. Thus, the end-user can select one of them and the end-user front-end adds the related service/device to the same location as the selected service/device. Finally, the feature model is updated. Fig 9 shows, as a representative example, how the end-user selects the *Alarm* service (1). This service represents a feature that has a Single Choice relationship. Then, a dialog is shown with all the devices that represent the related features (2). Then the end-user selects the *Siren* device and the feature model is updated activating the features *Alarm and Siren* (3).


The Security service in the Bedroom requires to set a device:

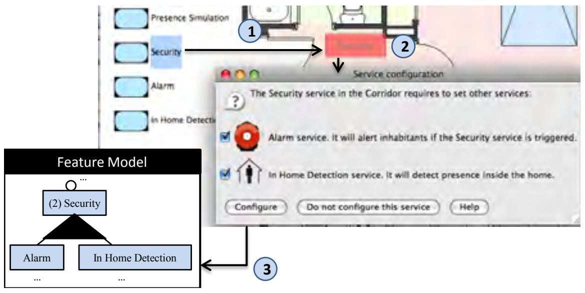
Siren Alarm. It will emit sounds.

Visual Alarm. It will set lights on a blinking state.

Silent Alarm. It will send you a text message.

Fig. 9. Applying patterns in an optional or single choice relationship

- If there is a **Multiple choice**, we use **Show some options and autocompletion**. When a feature A has a multiple relationship with other features, one or more of them has to be selected. In the end-user front-end, when the end-user selects that represents a feature with a multiple relationship, the end-user front-end shows a dialog with services/devices that represents the related features. Then, the end-user can select one of more of them and the end-user front-end adds them to same location where the previously selected service is located. In addition, the feature model is updated according to this selection. Fig 10 shows, as representative example, how the end-user selects the *Security* service (1). The feature that represents this service has a *Multiple Choice* relationship. Then, a dialog is shown with the devices that represent the related features (2). Afterwards, the end-user selects the *Alarm* Service and the *In Home Detection* services. Finally, the Feature Model is updated activating the features *Security, Alarm, and In Home Detection* (3).


The Security service in the Corridor requires to set other services:

Alarm service. It will alert inhabitants if the Security service is triggered.

In Home Detection service. It will detect presence inside the home.

Fig. 10. Applying patterns in a multiple choice relationship

V. Supporting technologies for the end-user oriented MDD-SPL

As we described in the previous section, our end-user tool uses the Feature Model to offer the catalog of available services/devices. This model is also used to save the end-user's configurations by activating/inactivating features. The feature model is specified using the MOSkitt Feature Modeller editor [31], which uses the technology provided by the Eclipse Modelling Platform [32].

Thus, in order to connect the end-user front-end with the feature model we have used the EMF Model Query framework [33]. EMF Query provides an API to construct and execute query statements. These query statements can be used for discovering and modifying model elements. Queries are first constructed with their query clauses and then they are ready to be executed.

There are two query statements available: SELECT and UPDATE. The SELECT statement provides querying without modification while the UPDATE statement provides querying with modification. The SELECT statement requires two clauses, a "FROM" and a "WHERE." The FROM clause describes the source of model elements where SELECT can iterate in order to derive results. The WHERE clause describes the criteria for a model element that matches. The condition provided to the WHERE clause falls under a specialized condition called an EObjectCondition which is specially designed to evaluate model elements.

We have implemented the interaction patterns described in the Subsection IV-B by using EMF Model Query. For instance, when the end-user selects the Alarm service, the tool checks the feature model for the selected feature. It also checks the relations with other features. In this case, the Alarm service is related with a single choice relation with three features (Silent Alarm, Siren and Visual Alarm). Thus, as the feature model relation is Single choice, the interaction patterns that are applied are (see previous section): (1) Show all options and (2) Autocompletion. Then, we need both to obtain the features related with the selected one in order to show all of them, and to update the selected feature and also the selected related feature.

Next, we show the query that we have implemented to obtain the child features of the single choice relationship by using EMF Model Query:

```
SELECT statement =
 new SELECT(
  new FROM(currentFeature.getContents()),
  new WHERE(new EObjectReferenceValueCondition(
  new EObjectTypeRelationCondition(
          FeatureModelPackagePackage.eINSTANCE
              .getFeatureRelationship()),
   FeatureModelPackagePackage.eINSTANCE.
                    getFeatureRelationship_From(),
  new EObjectInstanceCondition(SingleChoice))
  )
);
```

Given a feature (currentFeature) the select statement gets all the features related with the currentFeature with a single choice relationship (EObjectInstanceCondition(SingleChoice)). Then, these features are shown as service options on the dialog of Fig. 9.

Once the end-user chooses one of the presented options, the state of the selected feature and its related one is updated in the feature model from inactivated to activated. By contrast, if the end-user drops this kind of device into the trash, its state is updated to inactive.

VI. Conclusions and future work

Taking the advantage of current MDD techniques and an integrated SPL architecture, we have provided an interactive design tool that allows end-users (rather than engineers) to create tailored solutions that directly reflect their needs and expectations. In order to tackle this, we have presented an MDD-SPL approach based on Model Driven Development to develop smart home systems which is complemented with our end-user tool. We have also presented how the end-user tool gets and sets information of the feature model according to the end-user configurations. Furthermore, we have applied interaction patterns to the end-user tool which improve the user interface usability. Finally, we have presented the technology implementation for handling the feature model.

As future work, we plan to validate the end-user configurations in the end-user tool and assist end-users during the configuration process. To do this, we plan to use the feature model Analyser Framework [27]. Furthermore, we plan to involve end-users in the domain engineering phase. Our goal is the participation of end-users in the definition of new service configurations.

Acknowledgment

This work has been developed with the support of MEC under the project SESAMO TIN2007-62894 and cofinanced by FEDER, in the grants program FPI.

References

[1] M. K. Lee, S. Davidoff, J. Zimmerman, and A. K. Dey. Smart homes, families and control. In *Proceedings of Design & Emotion 2006*, 2006.

[2] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. *SPLC 2007*, pages 233–242, Sept. 2007.

[3] Javier Muñoz and Vicente Pelechano. Building a software factory for pervasive systems development. In *CAiSE*, pages 342–356, 2005.

[4] C. Cetina, J. Fons, and V. Pelechano. Applying software product lines to build autonomic pervasive systems. pages 117–126, Sept. 2008.

[5] Martijn van Welie and Hallvard Trætteberg. Interaction patterns in user interfaces. In *PLoP 2000*, pages 13–16, 2000.

[6] Mick P. Couper, Roger Tourangeau, Frederick G. Conrad, and Scott D. Crawford. What they see is what we get: response options for web surveys. *Soc. Sci. Comput. Rev.*, 22(1):111–127, 2004.

[7] Wilbert O. Galitz. *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[8] M. Anastasopoulos, T. Patzke, and M. Becker. Software product line technology for ambient intelligence applications. In *In Proc. Net.ObjectDays*, page 179Ű195, 2005.

[9] Jon O'Brien, Tom Rodden, Mark Rouncefield, and John Hughes. At home with the technology: an ethnographic study of a set-top-box trial. *ACM Trans. Comput.-Hum. Interact.*, 6(3):282–308, 1999.

[10] Rick Rabiser. Flexible and user-centered visualization support for product derivation. In *SPLC (2)*, pages 323–328, 2008.

[11] John Steinmetz. *Computers and Squeak as Environments for Learning*. 2000.

[12] David Canfield Smith, Allen Cypher, and Jim Spohrer. Kidsim: programming agents without a programming language. *Commun. ACM*, 37(7):54–67, 1994.

[13] Chin, Callaghan, and Clarke. An end-user programming paradigm for pervasive computing applications. *International Conference on Pervasive Services*, 0:325–328, 2006.

[14] Anind K. Dey, Raffay Hamid, Chris Beckmann, Ian Li, and Daniel Hsu. A cappella: programming by demonstration of context-aware applications. In *CHI '04*, pages 33–40, New York, USA, 2004.

[15] Jan Humble et al. Playing with the bits: User-configuration of ubiquitous domestic environments. In *UbiComp 2003*, 2003.

[16] Khai N. Truong, Elaine M. Huang, and Gregory D. Abowd. Camp: A magnetic poetry interface for end-user programming of capture applications for the home. In *Ubicomp 2004*, pages 143–160, 2004.

[17] Brad A. Myers, John F. Pane, and Andy Ko. Natural programming languages and environments. *Commun. ACM*, 47(9):47–52, September 2004.

[18] John Francis Pane. *A programming system for children that is designed for usability*. PhD thesis, Pittsburgh, PA, USA, 2002. Co-Chair-Myers,, Brad A. and Co-Chair-Garlan,, David.

[19] Henry Lieberman. Programming by example (introduction). *Commun. ACM*, 43(3):72–74, 2000.

[20] Andrew J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *CHI '04*, pages 151–158, 2004.

[21] Carnegie Mellon University. Alice. http://www.alice.org/index.php, 1 2009.

[22] The accord toolkit. http://www.sics.se/accord/toolkit.html, 1 2009.

[23] Javier Muñoz and Vicente Pelechano. Applying software factories to pervasive systems: A platform specific framework. In *ICEIS (3)*, pages 337–342, 2006.

[24] Javier Muñoz, Vicente Pelechano, and Carlos Cetina. Implementing a pervasive meeting room: A model driven approach. In *IWUC*, pages 13–20, 2006.

[25] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Comput. Networks*, 51(2):456–479, 2007.

[26] D. Benavides, Ruiz A. Cortés, and P. Trinidad. Automated reasoning on feature models. *CAiSE 2005*, 3520:491–503, 2005.

[27] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A.Jimenez. Fama framework. In *SPLC*, 2008.

[28] Marcos Didonet Del Fabro, Jean Bézivin, and Patrick Valduriez. Weaving models with the eclipse amw plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany*, 2006.

[29] Maria Francesca Costabile, Piero Mussio, Loredana Parasiliti Provenza, and Antonio Piccinno. End users as unwitting software developers. In *WEUSE '08*, pages 6–10, New York, USA, 2008.

[30] Henry Lieberman, Fabio Paternò, and Volker Wulf. *End User Development*. Springer, 2006.

[31] Moskitt feature modeller. www.pros.upv.es/labs/projects/mfm.

[32] Eclipse modelling framework. http://www.eclipse.org/modeling/.

[33] Emf model query. http://www.eclipse.org/modeling/emf/?project=query.

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 configurations. 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

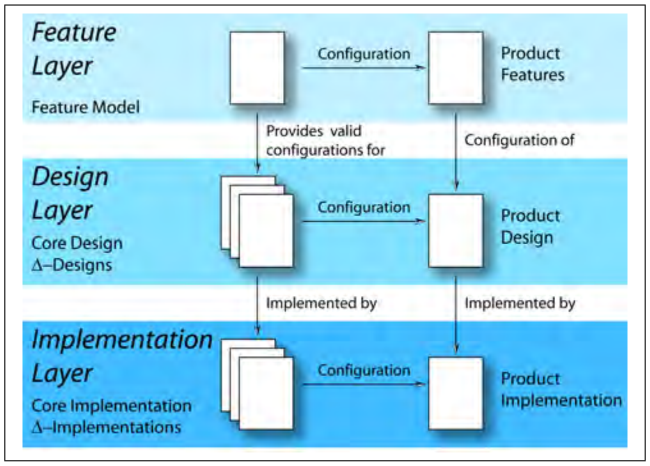


Figure 1. Model-based Automated Product Derivation

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

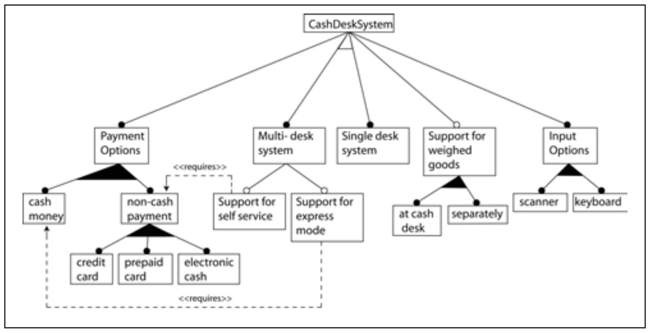


Figure 2. Feature Model for the CoCoME Software Product Line

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

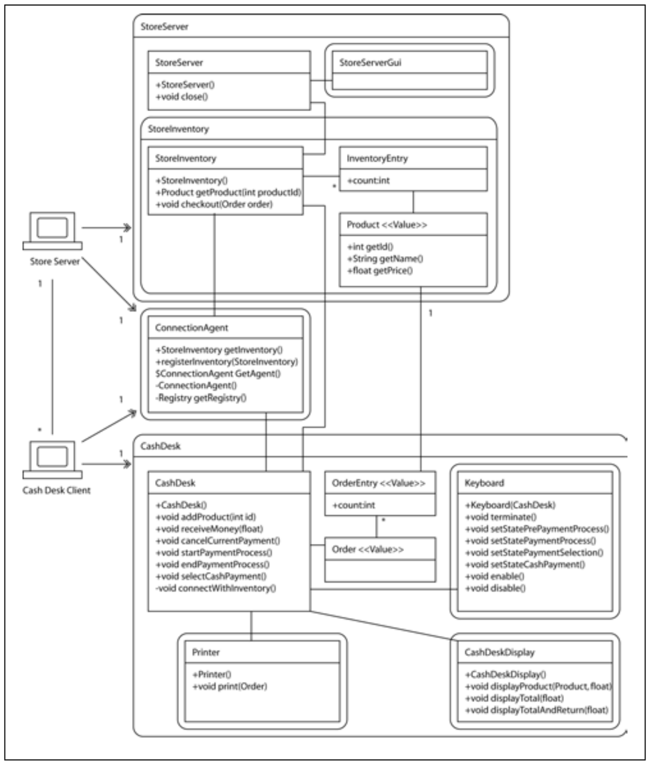

StoreServer
StoreServer
+StoreServer()
+void close()
StoreServerGui
StoreInventory
StoreInventory
+StoreInventory()
+Product getProduct(int productId)
+void checkout(Order order)
InventoryEntry
+count:int
Product <<Value>>
+int getId()
+String getName()
+float getPrice()
Store Server
ConnectionAgent
+StoreInventory getInventory()
+registerInventory(StoreInventory)
$ConnectionAgent GetAgent()
-ConnectionAgent()
-Registry getRegistry()
Cash Desk Client
CashDesk
CashDesk
+CashDesk()
+void addProduct(int id)
+void receiveMoney(float)
+void cancelCurrentPayment()
+void startPaymentProcess()
+void endPaymentProcess()
+void selectCashPayment()
-void connectWithInventory()
OrderEntry <<Value>>
+count:int
Order <<Value>>
Keyboard
+Keyboard(CashDesk)
+void terminate()
+void setStatePrePaymentProcess()
+void setStatePaymentProcess()
+void setStatePaymentSelection()
+void setStateCashPayment()
+void enable()
+void disable()
Printer
+Printer()
+void print(Order)
CashDeskDisplay
+CashDeskDisplay()
+void displayProduct(Product, float)
+void displayTotal(float)
+void displayTotalAndReturn(float)
1
1
1
1
1
1
*
*
*

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

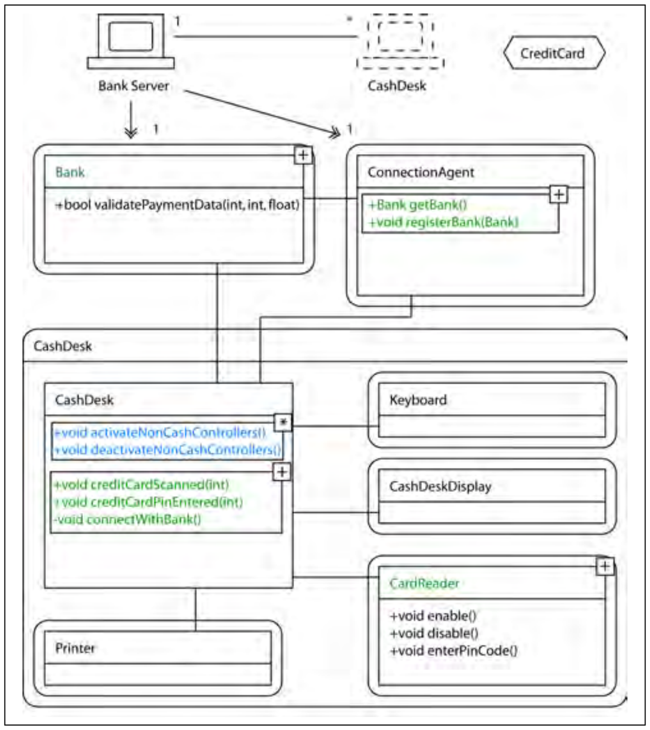


Figure 4. Δ-Design for Credit Card Payment

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

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.

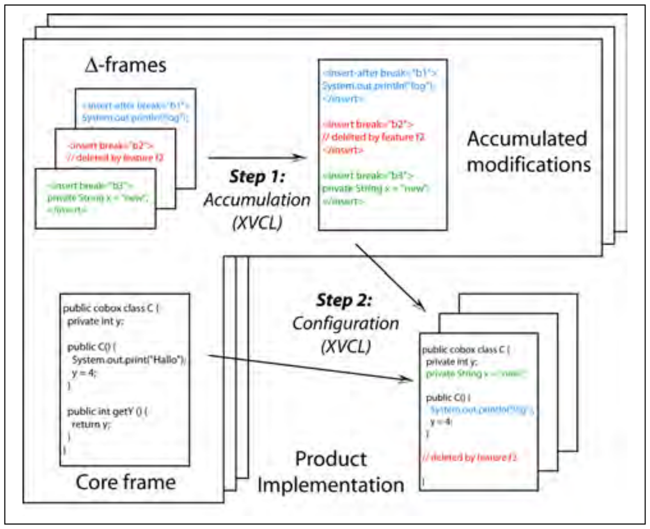


Figure 7. Automated Product Derivation using XVCL

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 Enginering (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*.

How to Configure a Configuration Management System – An Approach Based on Feature Modeling

Sebastian Wenzel
IBM Business Services GmbH
Wilhelm-Fay-Str. 30-34
Frankfurt, Germany
wenzel.sebastian@de.ibm.com

Thorsten Berger
University of Waterloo
200 University Ave West
Waterloo, ON, Canada
tberger@swen.uwaterloo.ca

Thomas Riechert
University of Leipzig
Johannisgasse 26
Leipzig, Germany,
riechert@informatik.uni-leipzig.de

***Abstract*—The accomplishment of an efficient IT service management is considered a significant success factor in large businesses. Configuration Management (CM) constitutes one of its core disciplines. Off-the-shelf CM systems support the maintenance of the IT by handling the lifecycle of so-called Configuration Items (CIs) and by establishing Change, Configuration and Release Management processes. However, due to the complexity of today's IT infrastructure in large companies, the tailoring of these systems based on concrete stakeholder requirements can become a laborious and error-prone task.**

We present an approach that enables the configuration of a CM system by leveraging variability management techniques stemming from product line engineering. The synthesis and configuration of a feature model is driven by the Common Data Model, a large domain-specific model that describes CIs and their relationships. We show how our feature-based approach can improve the tailoring of CM systems. Furthermore, we expand on its prototypical realization, elaborate on the integration into the requirements engineering process and discuss its applicability based on experiences obtained from a first evaluation.

IT service management; configuration management; feature modeling; requirements engineering

I. Introduction

During the past decades, the technological innovation of information technology has been the main driving force to achieve a higher level of efficiency and effectiveness within businesses [1]. However, the growing complexity of companies' IT environments has indicated a need for more comprehensive IT management support. One solution of tackling the growing complexity is the introduction of IT service management (ITSM) techniques. ITSM provides a process-centered view on the management of IT infrastructures and aims at assuring the quality of IT services.

One of the most important disciplines that ITSM comprises is Configuration Management (CM) which is responsible for keeping information about the managed IT infrastructure to be managed both up-to-date and accurate. According to Klosterboer [2], the implementation of CM is very difficult to accomplish. Many companies have problems with the realization of CM practices. Especially the tailoring and installation of the CM database and the establishment of change processes present some of the most complicated tasks. It is critical to design a concrete and accurate specification for the CM database that reflects all the data required for ITSM processes.

We were faced with the problem of configuring a CM database as part of an outsourcing project for a company that has to manage a large IT infrastructure with more than 2000 servers. The tailoring of the database, i.e. the creation of its concrete data model, was driven by requirements that had to be elicited from stakeholders. Additionally, the data model of the database had to conform to the *Common Data Model (CDM)*, a domain-specific model from IBM Tivoli that defines types of Configuration Items (CIs) and their relationships.

The manual and indirect tailoring of the database turned out to be very laborious and error-prone: First, the configuration knowledge is elicited indirectly via textual requirements from the customer. Second, the actual configuration has to be carried out by experts with significant knowledge about the database specification, the *CDM* elements and a considerable number of constraints.

In this context, we present a model-driven approach to creating a CM database specification that leverages Feature Modeling [3] techniques. It dynamically synthesizes a feature model that provides different levels of abstraction over the database specification, incorporates CI dependencies as constraints and supports a staged configuration process. In summary, it exposes the structure and configuration options of the database specification more explicitly and provides a more abstract view of it.

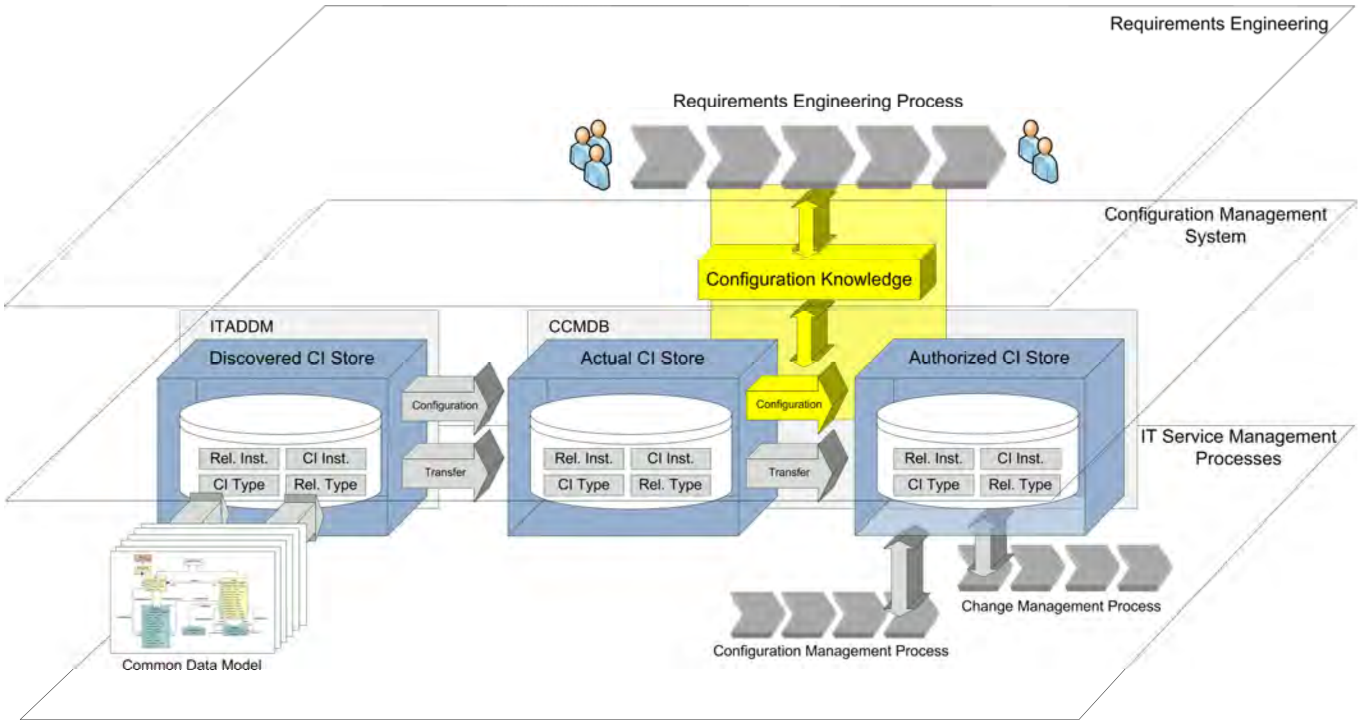


Figure 1. CM system and processes overview

The remainder of the paper is structured as follows: In section 2, we give an introduction to Configuration Management for IT services, describe the *Common Data Model* and portray our concrete problem context. Section 3 presents our approach that traces CM database tailoring back to a feature configuration problem. Section 4 expands on the prototypical realization of a tool relevant for our method and section 5 reports on the evaluation experiences gained. Finally, we discuss relevant conclusions and outline future work in section 6.

II. Configuration Management

CM basically denotes "the process responsible for maintaining information about Configuration Items required to deliver an IT Service, including their relationships. This information is managed throughout the lifecycle of the CI." [4]. For a more comprehensive introduction to CM, including a definition of Configuration Items, we refer to Alison et al. [5] and Lacy et al. [6].

A. System Architecture

Fig. 1 provides a high-level view on the realization of the CM system in our project context as well as the connection to the various service management processes. The system consists of two main parts: *ITADDM*[1] and *CCMDB*[2] [7]. *ITADDM* denotes the *Discovery System* responsible for discovering, collecting and storing information about the IT infrastructure. This information comprises CIs and their relationships and is saved in a database called *Discovered CI Store*.

However, not all the data that has been discovered by *ITADDM* is relevant for IT service management processes. Thus, the information is filtered and transferred into the *CCMDB*. The *CCMDB*, in turn, consists of two logical databases realized in one physical database. These logical databases are named *Actual CI Store* and *Authorized CI Store*. The former one just keeps a subset of the discovered data, but still contains sufficient information that is necessary for the CM system to operate correctly. This information is stored with a high level of detail and is necessary for root cause analysis, but not for the IT management itself. In contrast, the *Authorized CI Store* only keeps CIs and relationships that are subject to change and configuration management processes. This information is essential for a failure-free operation of the IT infrastructure.

Fig. 2 shows an example of how part of the data discovered by *ITADDM* is filtered for its usage in conjunction with IT service management processes. More precisely, the diagram shows parts of the *CI Stores'* specifications, which are sets of *CI Types* and their relationships. The *CI Types* themselves, their attributes and relationships are defined in IBM Tivoli's *Common Data Model*.

B. Common Data Model

The *Common Data Model*[3] is a domain-specific model that describes concepts in the CM domain. According to Tai et al. [8], *CDM* "provides consistent definitions for managed resources, business systems and processes, and other data, and

[1] IBM Tivoli Application Dependency Discovery Manager: http://www-01.ibm.com/software/tivoli/products/taddm/

[2] IBM Tivoli Change and Configuration Management Database: http://www-01.ibm.com/software/tivoli/products/ ccmdb/

[3] http://www.redbooks.ibm.com/redpapers/pdfs/redp4389.pdf

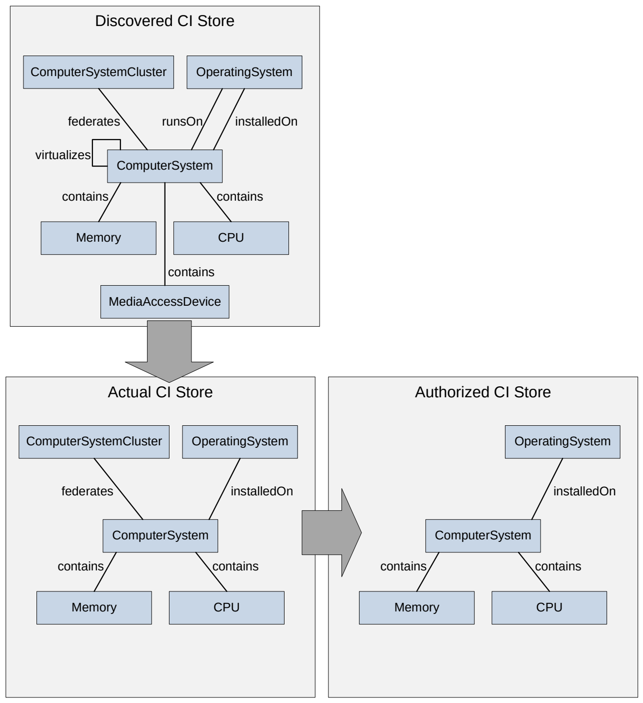


Figure 2. Filtering of CIs among the *CI Stores*

the relationships between those elements". Thus, it can be seen as a domain-specific language rather than just as a data model for the *CI Stores*. In fact, many CM tools from the IBM Tivoli family are built upon concepts as defined in the *CDM*.

Technically, it is modeled in UML2 and contains about 750 classes with attributes as well as 82 named association types (e.g. `contains`, `installedOn`, `virtualizes`). Three UML2 Profiles define stereotypes in order to specify technical tool and data mappings. *CDM* further introduces the notion of *Sections*, which categorize related classes. They are organized hierarchically and each of the 36 *Sections* corresponds to a concrete class diagram.

Classes that represent real-world CIs realize the interface `ConfigurationItem` and are subject to IT service management processes. Thus, they embody the main entities that are to be saved in the *CI Stores*. However, since administrative and meta-information also has to be stored in the *CI Stores*, all classes derived from `ModelObject` can be persisted in the databases. Furthermore, concrete relationships between classes are defined and named according to their corresponding association type. Altogether, almost 1600 unique relationships – defined as associations – exist in *CDM*.

C. *CI Store Specification*

In order to support the IT service management processes, the stores have to be tailored towards the stakeholders' requirements. Basically, this tailoring comprises the creation of a specification for the CM databases, i.e. for the *Actual* and the *Authorized CI Store*. A specification contains (1) a set of *CI Types* including meta/administrative information and (2) a set of relationships as defined by the *CDM*. Furthermore, a logical hierarchy is introduced, which is based on a specific relation between classes in *CDM*. This hierarchy is defined using a `Parent` attribute in classes, but each parent-child relation is further detailed by a corresponding association. Fig. 3 illustrates the mapping between a store specification and the *CDM*.

In this paper we focus, however, on the specification of the *Authorized CI Store*. Setting up the *Actual CI Store* is not addressed here since it is rather driven by technical aspects than by customer requirements. The mapping and transfer between *Discovered* and *Actual CI Store* is realized by predefined adaptors with the option to define the hierarchy depth.

D. *Authorized CI Store*

The current process of creating a specification for the *Authorized CI Store* can be characterized as follows:

Elicitation of requirements from the customer: Based on the current specification of the *Actual CI Store*, requirements reflecting the necessary *CI Types* and relations have to be elicited from the stakeholder. Our project, for example, comprised more than 700 requirements [9].

Analysis of requirements: CIs, meta/administrative information and relationships that are to be transferred from the *Actual* to the *Authorized CI Store* have to be identified on the basis of the elicited requirements and the *CDM*. In practice, requirements are currently mapped to *CDM* elements in Microsoft Excel spreadsheets.

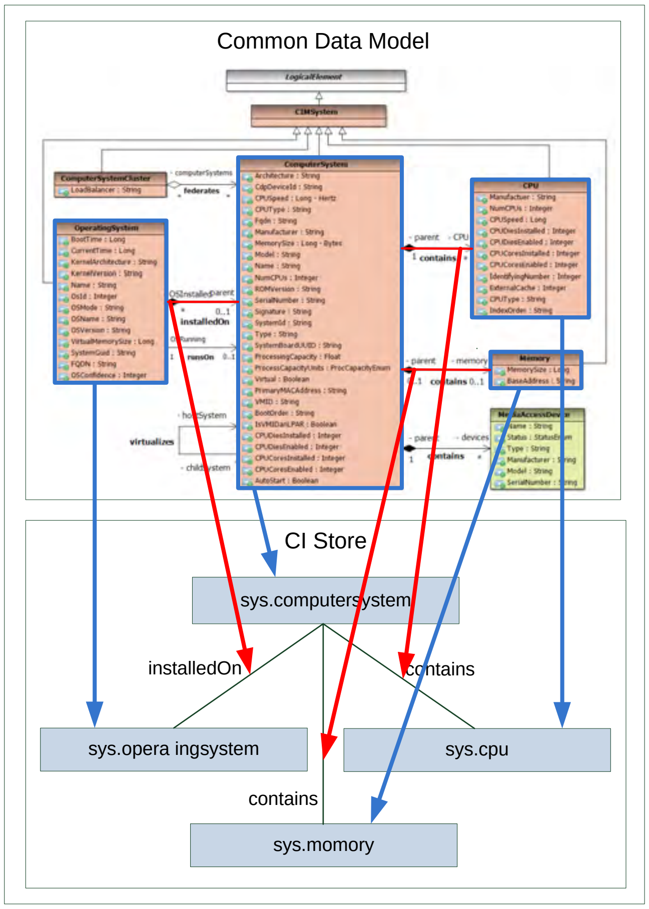


Figure 3. Mapping between *CDM* and *CI Store* specifications

Applying the specification: Finally, the specification has to be applied to the *Authorized CI Store* by entering all elements into a web configuration interface. Furthermore, the former hierarchy of the *Actual CI Store* has to be retained or recreated.

In its current form, the process turns out to be quite ineffective for the following reasons: First, profound knowledge about possible *CI Types* and relationships is expected from the stakeholders. Second, available elements are limited by the current specification of the *Actual CI Store*. Third, consistency between *CI Types* and relationships is difficult to maintain. Furthermore, terminology and translation issues concerning the textual requirements occur.

III. Feature Model Synthesis and Configuration

In order to bridge the gap between the (1) *Actual CI Store* specification, the definitions in the (2) *CDM* and the implicit (3) *configuration knowledge* of the stakeholders, we introduce an approach based on Feature Modeling and Feature Model Configuration [10,11] techniques as known from Software Product Line Engineering [12,13].

We try to reduce the disadvantage of the current method by providing a simplified and more coherent view on the *Actual CI Store specification* in form of a feature model. This model provides a higher level of abstraction for the selection of relevant *CI Types* and relations that are essential for the stakeholders. The goal is to obtain a specification for the *Authorized CI Store*.

Our approach (cf. Fig. 4) consists of three main steps:

- Feature Model Synthesis
- Feature Model Configuration
- *Authorized CI Store* Creation

The approach facilitates the configuration of the feature model on different levels of abstraction. On the highest level, the presented view is intended to be simpler and easier to understand for stakeholders without specific knowledge about the underlying *CDM*.

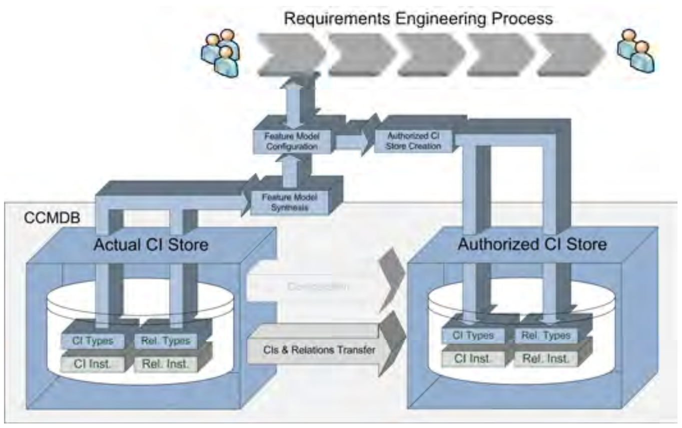


Figure 4. Feature model synthesis and configuration steps

A. *Feature Model Synthesis*

The first step of our approach deals with the dynamic creation of a feature model. The model is based on the current specification of the *Actual CI Store* and allows an adjustable representation of the prospective *Authorized CI Store*. We introduce four types of features:

- *Diagram Concept features*: root feature describing the underlying logical data model (i.e. the scope of the feature model).
- *CDM Section features*: describing the highest abstraction level of the *CDM - Sections*.
- *CI Type features*: representing *CI Types* contained in the logical data model.
- *CI Relation features*: representing relations between *CI Types*.

The feature model is built in three stages. In each stage features of different types are added to the model. All of them are optional, we didn't need to introduce mandatory features or mutual exclusions. An example of the feature model levels, created by the described procedure, is presented in Fig. 5. The synthesis stages are as follows:

The first stage consists of two steps: the creation of the *Diagram Concept feature* (e.g. *Actual CI Store*) and the creation of *CDM Section features* (e.g. `Administration Section` or `ComputerSystem Section`). These features are either child features of the *Diagram Concept feature* or of other *CDM Section features*. This stage of feature model synthesis is initially executed once for all projects.

The second stage of the synthesis comprises the creation of *CI Type features* corresponding to *CDM Sections*. The parent feature of these features is a *CDM Section feature*. This stage is automatically executed on the basis of the *CDM Section – CI Type* mapping and the *Actual CI Store* structure. For instance, the *CI Types* `CPU` and `ComputerSystem` belong to the *CDM* Section `ComputerSystem Section` and are part of the *Actual*

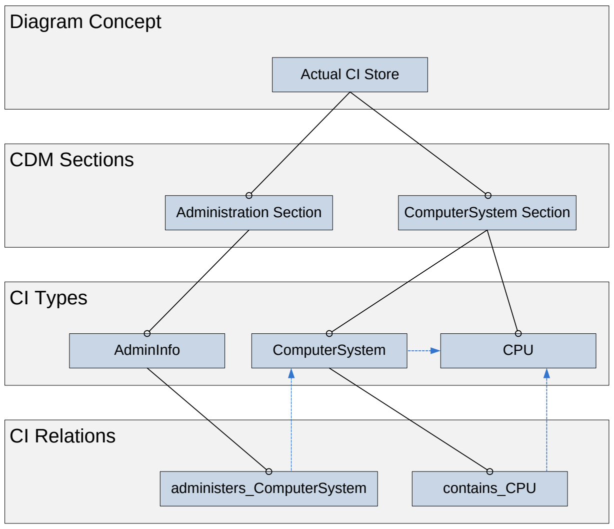


Figure 5. Levels of the feature model

CI Store; thus, they are added to the feature model as children of the `ComputerSystem Section` feature. If the added feature represents a *CI Type* which is not labeled in the *Actual CI Store* as top-level, a constraint pointing to the feature of its parent *CI Type* in the logical *Actual CI Store* hierarchy is added to the feature model.

The third stage of the synthesis creates *CI Relation features*. This stage is automatically executed on the basis of the *Actual CI Store* structure. Those CI relations are added to the feature model, for which the source *CI Type* and the target *CI Type* exist in the feature model. They are added to the model as children of the source *CI Type* feature. Furthermore, for each *CI Relation feature*, a constraint pointing to the target *CI Type feature* is added to the feature model.

B. *Feature Model Configuration*

The second step of the feature-based approach comprises a kind of staged configuration of the synthesized feature model. This configuration is performed by the stakeholders in order to select features directly and, thus, to omit or at least reduce the error-prone elicitation of requirements. We also leverage the choice propagation functionality in feature model tools for the purpose of assuring relationships, which have been added as extra constraints to the feature tree.

In summary, this step extends the current requirements engineering that is carried out for gaining configuration knowledge from stakeholders (cf. Fig. 1 and Fig. 4).

The configuration of the feature model is executed in three stages. The initial feature model is created in the first Feature Model Synthesis stage. In the **first configuration stage** the *CDM Sections* relevant for the stakeholder are selected. After that, the second feature synthesis stage is performed and the *CI Type features* corresponding to the selected *CDM Sections* are loaded. This allows the execution of the **second configuration stage** in which the stakeholders select the required *CI Types*. Based on the selected *CI Types*, the third feature model synthesis stage is executed and *CI Relation features* are added to the feature model. The **third configuration stage** is performed on the basis of the CI relations added in the third synthesis stage. The CI relations necessary for the stakeholder's IT infrastructure are selected, resulting in the final configuration of the feature model. This configuration is the basis for the specification of the *Authorized CI Store*.

C. *Authorized CI Store Creation*

The last step of our feature-based approach constitutes the creation of the *Authorized CI Store* specification. This specification is generated on the basis of the final configuration of the feature model (see Fig. 6). The *Authorized CI Store* specification is subdivided into two parts: a list of *CI Types* selected by the stakeholders and a list of selected CI relations between those selected *CI Types*. These specification lists are saved in database-specific XML format. Based on these XML files, the *Authorized CI Store* logical hierarchy is created in the CM system.

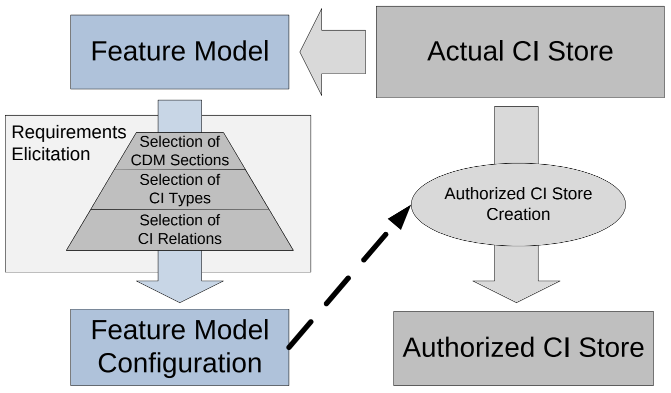

Feature Model
Actual CI Store
Requirements Elicitation
Selection of CDM Sections
Selection of CI Types
Selection of CI Relations
Authorized CI Store Creation
Feature Model Configuration
Authorized CI Store

Figure 6. Requirements elicitation-based feature model configuration

IV. PROTOTYPICAL REALIZATION

We have realized our approach as an Eclipse plug-in, since we wanted to be able to embed it with other tools from IBM Tivoli and since we chose to integrate with FMP[4] [14] as a Feature Modeling tool. FMP turned out to be the most appropriate one for our purpose. It is available as Open Source software, supports basic Feature Modeling with extra constraints, staged configuration and choice propagation. Cardinalities are also supported in FMP, but were not necessary for our approach.

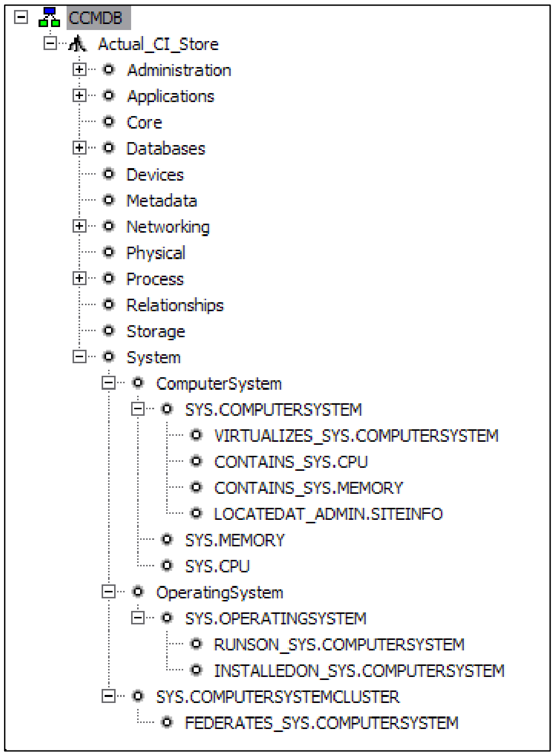
In summary, our plug-in extends FMP, realizes the feature model synthesis and staged configuration as well as it provides adapters for the *Actual CI Store* in order to obtain the current specification.

As described in section 3, the synthesis procedure creates a feature model in FMP by leveraging the structure of *CDM Sections* and loading the current *Actual CI Store* specification. We load subsections just on demand since we faced performance issues[5] when creating the whole feature model from a large *Actual CI Store* in one step. Our plug-in adds relationships as subfeatures and adds binary constraints in FMP in order to support choice propagation. Since there exists another logical hierarchy between CIs (cf. section 2.3), additional constraints representing it are introduced into the feature model. For further implementation details such as naming rules, feature ID definition for traceability reasons, or constraint realization, we refer to [15].

Fig. 7 illustrates the feature model view, especially with the `ComputerSystem` and `OperatingSystem` sections. Fig. 8 shows a list of constraints of the feature model presented on Fig. 7. For instance, constraints between the features `SYS.COMPUTERSYSTEM` and `SYS.OPERATINGSYSTEM` and between relations and whose target *CI Types*.

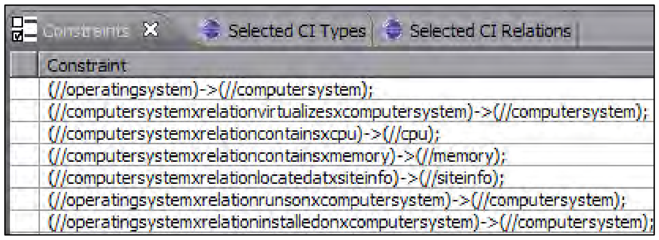
[4] Feature Modeling Plug-in: http://fmp.sf.net

[5] These are known issues owed to FMP's meta-modeling and just-in-time reasoning capabilities.



```
CCMDB
  Actual_CI_Store
    Administration
    Applications
    Core
    Databases
    Devices
    Metadata
    Networking
    Physical
    Process
    Relationships
    Storage
    System
      ComputerSystem
        SYS.COMPUTERSYSTEM
          VIRTUALIZES_SYS.COMPUTERSYSTEM
          CONTAINS_SYS.CPU
          CONTAINS_SYS.MEMORY
          LOCATEDAT_ADMIN.SITEINFO
        SYS.MEMORY
        SYS.CPU
      OperatingSystem
        SYS.OPERATINGSYSTEM
          RUNSON_SYS.COMPUTERSYSTEM
          INSTALLEDON_SYS.COMPUTERSYSTEM
      SYS.COMPUTERSYSTEMCLUSTER
        FEDERATES_SYS.COMPUTERSYSTEM
```

Figure 7. Feature model example



Constraint
(//operatingsystem)->(//computersystem);
(//computersystemxrelationvirtualizesxcomputersystem)->(//computersystem);
(//computersystemxrelationcontainsxcpu)->(//cpu);
(//computersystemxrelationcontainsxmemory)->(//memory);
(//computersystemxrelationlocatedatxsiteinfo)->(//siteinfo);
(//operatingsystemxrelationrunsonxcomputersystem)->(//computersystem);
(//operatingsystemxrelationinstalledonxcomputersystem)->(//computersystem);

Figure 8. Feature constraints

V. Evaluation

We evaluated our approach and the realized prototype in a small-scale setup with some colleagues. Although they did not represent stakeholders, they were familiar with customer projects. Their experience with *CDM* and the CM system ranged from deep to no experience at all with *CDM*.

Based on the goal of our work, we wanted to know (1) if the synthesized feature model provides a simplified view on the *CDM*-based *Actual CI Store* specification, (2) if our approach speeds up creating the specification and (3) how the tool would be accepted by stakeholders.

Accordingly, we gave a quick introduction into the approach and the tool. Thereafter, the participants performed a test scenario and created an *Authorized CI Store* specification. Finally, we asked them to fill out a questionnaire with nine questions.

We received very positive answers from the participants (for details cf. [15]): (1) The tree-based navigation and the support of constraints within the configured feature model were regarded as a significant advantage. (2) All participants also mentioned the time-saving potential. However, some of them also pointed out that time saving depends on the project size, i.e. the difference could be marginal for smaller projects. (3) Furthermore, participants agreed on the potential to increase customer acceptance, since less knowledge about *CDM* is necessary when using the tool. However, experts might miss some additional information that is intentionally omitted in the feature model.

In summary, the feature-based approach met with favor and appreciation participants of the evaluation. Especially the convenience and the focus on the stakeholder's interests and goals were emphasized very positively.

VI. Related Work

Although our approach is – to a certain degree – specific to the *CDM*, we depict some work that, in a broader sense, deals with variability in data models or data specifications by using Feature Modeling techniques.

Usually, feature models are used in various kinds of domain analysis. However, there is some work that uses feature models to provide a tree-oriented-view on fine-grained data with many relationships. Czarnecki et al. [16] elaborate on the expressiveness of feature models compared to rich ontology modeling techniques. In their work, they also provide a case study that synthesizes a feature model from a domain-specific ontology, i.e. they accomplish a more abstract view on domain data.

Barthold et al. [17] address the problem of variability in data models that appears in conjunction with software variability. They propose an approach to represent and manage data variability in entity models. Their approach is based on adapters that provide a specific view on the database, i.e. they, for example, omit entities or relations that are not relevant for a certain feature.

Some work that deals with mappings between UML diagrams and feature models comprises for example the following: Braganca and Macada [18] provide a mapping between features and the elements of Use Case diagrams. They establish a model-driven approach to deriving a concrete Use Case diagram that represents one product of a product line based on the feature configuration. Furthermore, Czarnecki and Antkiewicz [19] treat class and activity diagrams as templates containing variability in order to derive concrete model instances. They also deal with checking the consistency of derived UML diagrams.

VII. Conclusions and Future Work

In this work, we have developed a feature-based approach to creating a data specification for a *CI Store*. We dynamically synthesize a feature model that represents such specifications on a higher level of abstraction and provides a simplified view that is more stakeholder-oriented. This model is configured in three stages in order to obtain a concrete *CI Store* specification. The aim of our approach was to reduce the gap between stakeholder's implicit configuration knowledge and the complex *Common Data Model's* definitions of CIs and relationships.

More precisely, we defined a mapping between features and *CDM* elements that exploits structural characteristics in order to obtain a hierarchical feature tree. Further *CDM* relationships are incorporated as extra constraints of the feature tree. We have realized the approach as a tool prototype and have performed a first, small-scaled evaluation.

However, there is definitely room for improvement in this field and several enhancements to the method are possible. The current focus was on providing a general view for all stakeholders on the complex *CDM*-based specifications. Stakeholder may be even more enabled to configure this complex data model by using hierarchically structured feature models that are tailored towards particular groups. View integration and derivation with feature models, as proposed in [16], could provide interesting opportunities. Concerning the actual configuration by the stakeholders, an increase of the number of stages would also be possible. Furthermore, the synthesized feature model could be extended with additional features that reflect supplementary meta information, as requested by some evaluation participants.

Another reason for extending the approach lies in the conceivable evolution of *CI Stores*. When IT service management processes change, the modifications have to be reflected in the *CI Store* specification as well.

ACKNOWLEDGMENT

We would like to thank our colleagues from the IBM Service Management Department, especially Claudia Dahl and Jürgen Pfaffenberger for supporting this work, as well as the evaluators of the prototype.

REFERENCES

[1] M. Khosrow-Pour and M. Khosrowpour, Cases on Information Technology And Business Process Reengineering, IGI Publishing, 2006.

[2] L. Klosterboer, Implementing ITIL Configuration Management, IBM Press, 2008.

[3] K. Czarnecki and U.W. Eisenecker, Generative Programming. Methods, Tools and Applications: Methods, Techniques and Applications, Addison-Wesley Longman, 2000.

[4] Office of Government Commerce, "ITIL V3 Glossary: Glossary of Terms, Definitions and Acronyms," 2009.

[5] C. Alison, A. Hanna, C. Rudd, I. Macfarlane, J. Windebank, and S. Rance, An Introductory Overview of ITIL V3, The UK Chapter of the itSMF, 2007.

[6] S. Lacy and I. Macfarlane, Service Transition, The Stationery Office Ltd, 2007.

[7] H. Madduri, S.S.B. Shi, R. Baker, N. Ayachitula, L. Shwartz, M. Surendra, C. Corley, M. Benantar, and S. Patel, "A configuration management database architecture in support of IBM service management," IBM Syst. J., vol. 46, 2007, pp. 441-457.

[8] L. Tai, R. Baker, E. Edmiston, and B. Jeffcoat, IBM Tivoli Common Data Model: Guide to Best Practices, IBM International Technical Support Organization: http://www.redbooks.ibm.com/abstracts/redp4389.html (2009.06.10), 2008.

[9] C. Dahl, Configuration Management: System Requirements Specification, IBM Corp., 2007.

[10] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration Using Feature Models," Software Product Lines, 2004, pp. 266-283.

[11] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson, and C.U.P.P.S.E. INST, Feature-oriented domain analysis (FODA) feasibility study, The Institute, 1990.

[12] K. Pohl, G. Böckle, and F.V.D. Linden, Software Product Line Engineering. Foundations, Principles, and Techniques., Springer, Berlin, 2005.

[13] M. Sinnema and S. Deelstra, "Classifying variability modeling techniques," Inf. Softw. Technol., vol. 49, 2007, pp. 717-739.

[14] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: Feature Modeling Plug-in for Eclipse," In proceedings of the Workshop on Eclipse Technology eXchange, 2004, pp. 67-72.

[15] S. Wenzel, "How to Configure a Configuration Management Database – An Approach Based on Feature Modeling," Diploma Thesis, University Leipzig, 2009.

[16] K. Czarnecki, C.H.P. Kim, and K.T. Kalleberg, "Feature Models are Views on Ontologies," Proceedings of the 10th International on Software Product Line Conference, IEEE Computer Society, 2006, pp. 41-51.

[17] J. Bartholdt, R. Oberhauser, and A. Rytina, "An Approach to Addressing Entity Model Variability within Software Product Lines," Software Engineering Advances, 2008. ICSEA '08. The Third International Conference on, 2008, pp. 465-471.

[18] A. Braganca and R.J. Machado, "Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines," 11th International Software Product Line Conference, 2007. SPLC 2007, 2007, pp. 3-12.

[19] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," Lecture Notes in Computer Science, vol. 3676, 2005, p. 422.

Interactive Configuration of Embedded Systems Product Lines

Goetz Botterweck
Lero – The Irish Software Engineering Research Centre
University of Limerick
Limerick, Ireland
Email: goetz.botterweck@lero.ie

Andreas Polzer and Stefan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
Email: {polzer, kowalewski}@embedded.rwth-aachen.de

***Abstract*—This paper addresses product configuration and product derivation in product lines of embedded systems. We show how domain-specific languages (DSLs), which are used to describe the implementation of the product, can be translated into configurable models with formal semantics. This facilitates, tool support during configuration including (1) side-by-side visualization of features and corresponding implementation components, (2) automated reasoning to calculate consequences of the user's configuration decisions and (3) visual explanations for automatically calculated consequences. In addition, we discuss (4) how a completed configuration can be turned into a product-specific model in the domain-specific language, using negative variability and subsequent pruning of the implementation model.**

The approach is demonstrated for product lines of embedded systems using Simulink as an domain-specific language for the model-based engineering of embedded systems. We report on first evaluation results with a product line of parking assistant applications, including experimentation on a rapid prototyping platform with a 1:5 model car.

I. Introduction

Many approaches in Software Product Lines (SPL) structure the applied models into two areas (see figure 1): A model describing the available choices, e.g., a feature or variability model A_d and, one or more implementation models, which describe how these choices are implemented C_d. Usually these two types of models are mapped onto each other B to support further techniques.

During *Product Configuration* the user (i.e., a customer or an application engineer supporting him) decides which of the available product options to choose. In *Product Derivation*, he generates or assembles the product implementation that corresponds to these configuration decisions.

There are well-known techniques and tools for the interactive configuration of feature models, for instance in commercial tools such as *pure::variants* [1] or our earlier work on interactive configuration with formal semantics [2], [3].

Interestingly, during product configuration the developer typically configures the feature model A_d *only* – even though the mappings between the feature model and other SPL models are available B. Interaction with the implementation models C_d is usually *not* provided at this stage.

While there might be good reasons to abstract from the implementation deliberately (e.g., to hide complexity), the inclusion of other models in the interactive configuration process can provide additional benefits:

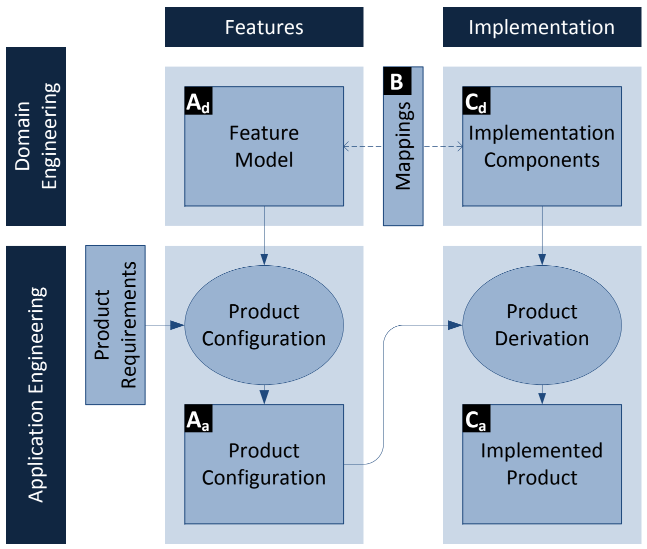


Fig. 1. Simplified Framework for Software Product Line Engineering.

1) The user can see (visual representations of) dependencies between features and the related elements in other models (e.g., an edge representing that the feature f_1 requires the component c_1).
2) When making configuration decisions (in the feature model) the user can immediately see consequences in related models (e.g., after the user selects the feature f_1, the tool automatically selects the component c_1).
3) Moreover, it is possible to apply configuration decisions in the implementation model first (e.g., indicating that the implementation component c_1 will not be available) and derive implications for the feature model from that (f_1 is no longer an option).

In this paper, we address this side-by-side configuration by integrating the implementation model into the configuration process. This allows us to provide the functionality sketched above. To further motivate our research, we will first use a sample case from product lines of embedded systems (section II). We will then explain our approach (sections III

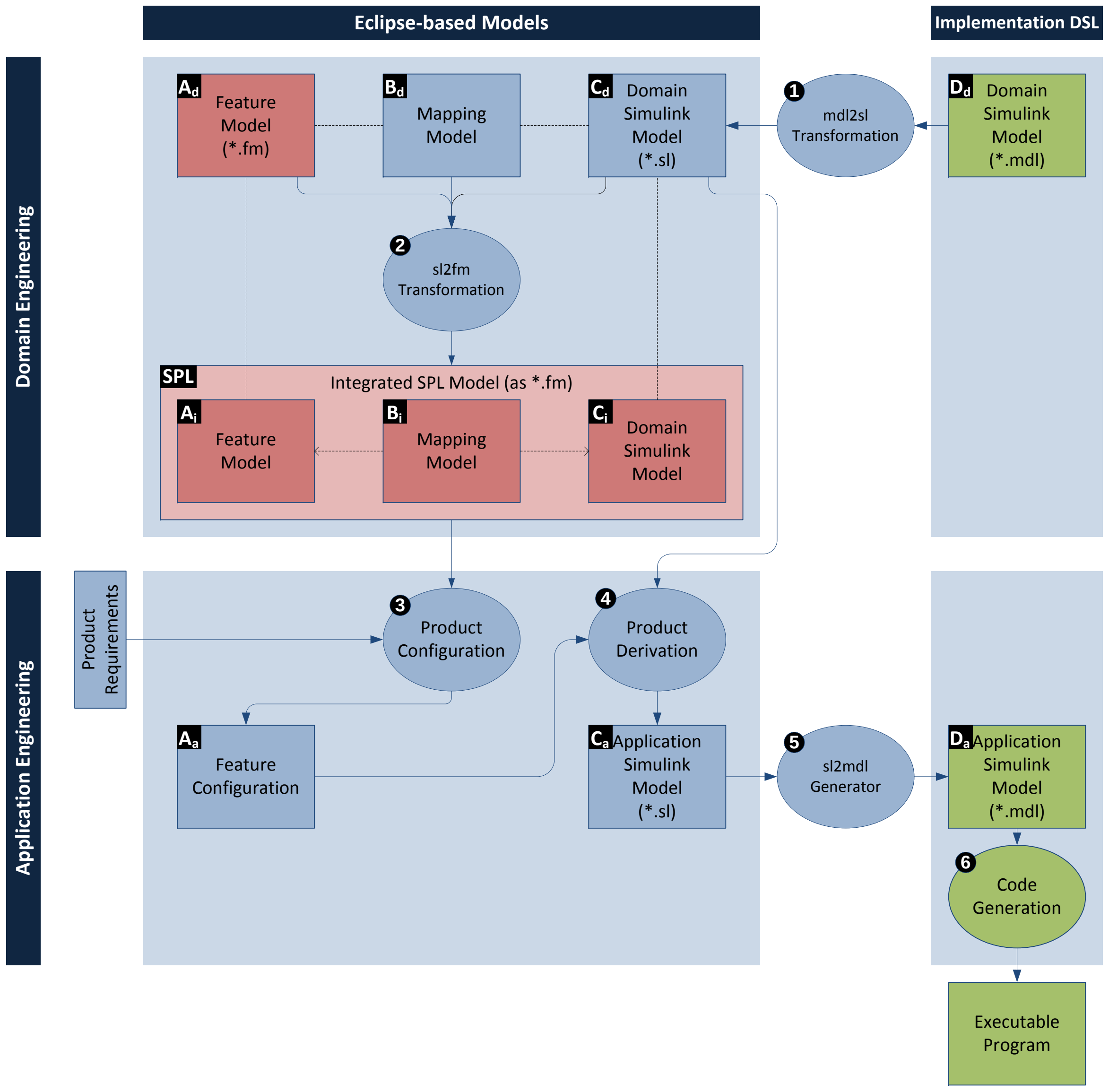


Fig. 2. Overview of our approach.

to VI). The paper concludes with a discussion (section VII), an overview of related work (section VIII) and some final remarks.

II. Sample Case: Embedded Systems in Control Engineering

As a sample case for this paper, we want to apply Software Product Line techniques to the domain of control engineering. Such control system are typically developed using model-based tools like Matlab/Simulink. They can be considered as (or implemented as) a special form of embedded systems.

The task of a *controller* is to influence an environment with *actuators* to achieve a certain behavior. To this end, the controller gathers information of the environment using *sensors*. With the information provided by the sensors the controller uses the actuators to influence the controlled environment. In many cases, the part of the environment that is the object of control (observed by the sensors, influenced by the actuators) is called the *plant*. The whole system (consisting of sensors, controller, actuators, and the plant) is called a *control loop* [4].

When developing the code for such a controller, the main requirements are the reaction time, the input-output stability and the control error. The behavior of the control loop depends on both the controller and the plant. To understand the behavior of the plant and the controller, models of both are developed in a first step. These models are improved using the results of simulation.

If the desired behavior is achieved, a next development step is performed. The model of the controller will be translated to source code and executed using a prototyping hardware. During this development step either a prototyping plant or a real-time model of the plant is influenced by the controller. In this development timing requirements can be observed and tested.

During the third development step the controller code is executed on the real product hardware. This is the first time the real plant (and not the simulated one) is tested with the controller. Cost and safety issues are the main reasons for the late testing with the real plant. One important task during this stage is to optimize the controller. To this end, the controller has parameters that can adopted until the control loop finally meets the requirements.

To built such systems, model-based design is a common engineering practice. Simulink is a very well-known example of a domain-specific modeling language for embedded systems, including corresponding tools. Using such development frameworks is one way to tackle the increasing complexity.

While this is already a nice foundation, in an industrial context we require additional techniques that help us to build whole product lines of such systems. To this end, model-based design techniques for embedded systems are extended with mechanisms for variability and model-driven product derivation.

We discussed some concepts and techniques for this in [5], [6] where we extend domain specific implementation models with variability. In this paper we focus on how feature models and the related implementation models can be combined to support their integrated, interactive configuration.

III. Overview of our Approach

Before we explain the details of our approach, we first want to give an overview as an orientation for the reader, see figure 2. Similar to common SPL Engineering methods our approach can be structured into *Domain Engineering* (upper layer) and *Application Engineering* (lower layer).

The overall goal of this process is to turn a product line implementation $\boxed{D_d}$ (upper right corner of figure 2) into an product-specific implementation $\boxed{D_a}$ (lower right corner) and finally an executable program.

To support common techniques and processes from Embedded Systems Engineering, we integrated our approach with the domain specific language *Simulink*. Hence, the chain of processes ❶ to ❻ begins in the Simulink world (right-hand side of figure 2) moves over (❶) to the Eclipse-based Models (left-hand side), which are configured and processed. Finally, by code generation (❺) we return to the implementation DSL. In the following sections we will now discuss these processes in more detail.

IV. Domain Engineering

For the context of this paper, we will assume that most processes, which are necessary to create the product line artefacts ($\boxed{A_d}$ to $\boxed{D_d}$) have already been performed. See [5] for more details.

Those processes that are of interest here, start off with the *mdl2sl transformation* ❶, which converts the implementation given in Simulink's native .mdl format $\boxed{D_d}$ into an corresponding Eclipse-based implementation model $\boxed{C_d}$. Subsequently, we are able to map this model to the feature model and perform further processing with Eclipse-based frameworks, such as the numerous frameworks from the Eclipse Modeling Project (EMP) [7] or openArchitectureWare (oAW) [8].

To enable the integrated configuration of feature and implementation models, we transform ❷ these models and the mappings between them into an integrated SPL model $\boxed{\text{SPL}}$. The basic idea is to represent all configurable parts of the product line (feature selected? component present?) as one large feature tree, where different subtree represent different SPL models. So, for instance, we can have one subtree with the real feature model $\boxed{A_i}$ and one subtree representing the configuration status of components $\boxed{C_i}$, as well as mappings and between them $\boxed{B_i}$. This enables us to interactively configure the whole product line within one integrated model.

This translation is realized by an model transformation in ATL (Atlas Transformation Language) [9]. It applies an semantic interpretation of the domain-specific concepts in the Simulink model, translating them into feature model elements, which make up the integrated SPL model $\boxed{\text{SPL}}$. Some rules for translation are shown in Table I for the Simulink model (translating from $\boxed{C_d}$ to $\boxed{C_i}$) and Table II for the mapping model (translating from $\boxed{B_d}$ to $\boxed{B_i}$).

Simulink $\boxed{C_d}$ (concepts in the meta-model)	Representation within the integrated SPL model $\boxed{C_i}$
Simulink Model m	mandatory feature $f(m)$
System s	optional feature $f(s)$
Block b	optional feature $f(b)$
contained blocks/systems in blocks/systems	subfeatures
Line from block a to block b	$f(b)$ requires $f(a)$ but not vice-versa

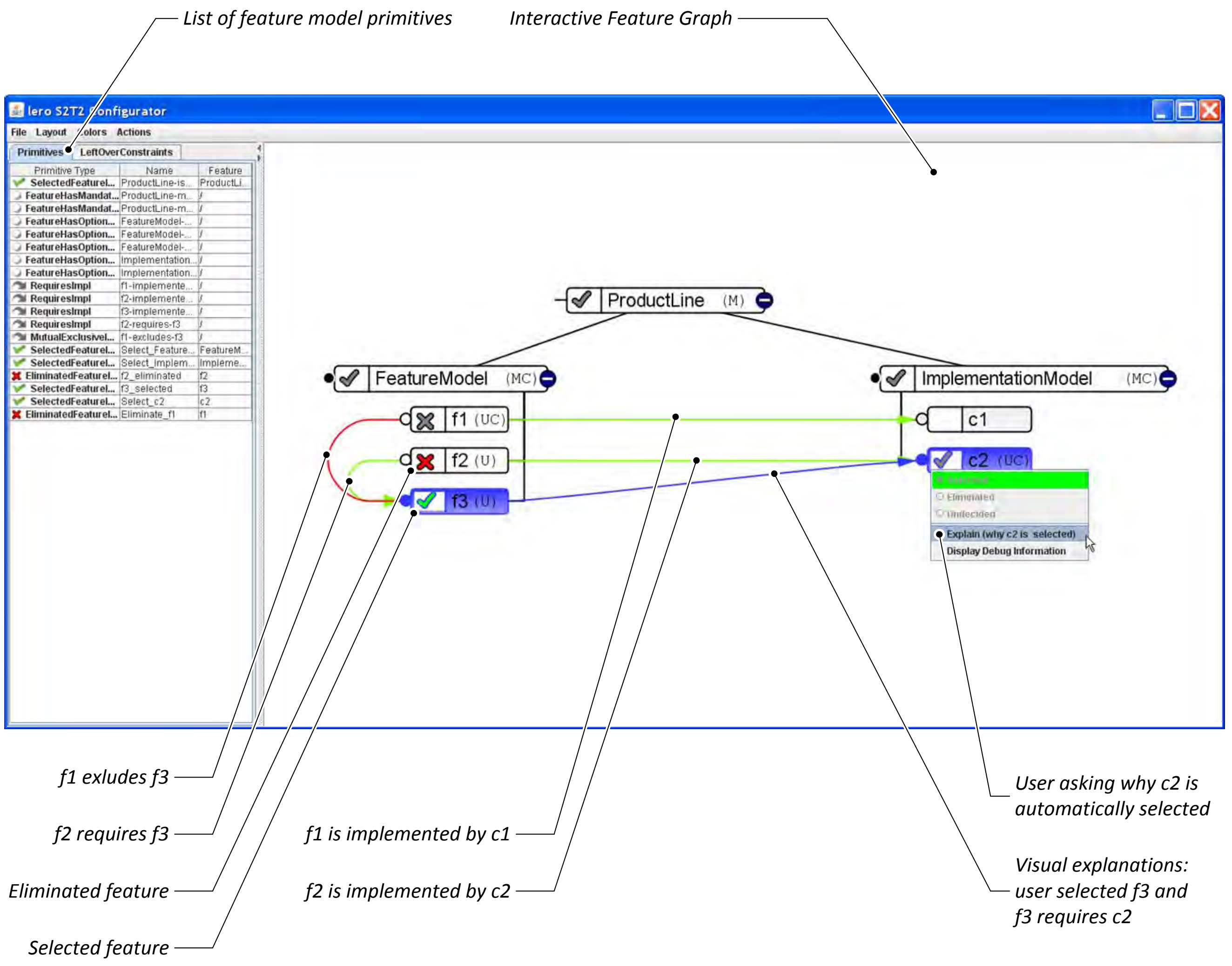
TABLE I
Translation from Simulink to the integrated SPL model

Mapping $\boxed{B_d}$ (concepts in the meta-model)	Representation within the integrated SPL model $\boxed{B_i}$
Feature f mapped to component c	$f(f)\ requires\ f(c)$

TABLE II
Translation from mapping model to the integrated SPL model

The elements created in the target model are fine grained elements of a feature model, which we call *feature model primitives*. Examples for such feature model primitives are f_1-*hasOptionalSubfeature*-$f2$, f_3-*requires*-f_4 or $f1$-*hasBeenSelected*. These primitives have clearly defined semantics, including the corresponding behavior of our `S2T2 Configurator` tool during interactive configuration. These semantics are given by further translation of the feature model primitives into propositional logic. For instance, f_3-*requires*-f_4 would be translated into $\neg f_3 \vee f_4$. Details of this translation can be found in [2].

In summary, the transformations provide the following result: We now have (1) an integrated model presenting features,



Primitive Type	Name	Feature
✓ SelectedFeatureI...	ProductLine-is...	ProductLi...
FeatureHasMandat...	ProductLine-m...	/
FeatureHasMandat...	ProductLine-m...	/
FeatureHasOption...	FeatureModel-...	/
FeatureHasOption...	FeatureModel-...	/
FeatureHasOption...	FeatureModel-...	/
FeatureHasOption...	Implementation...	/
FeatureHasOption...	Implementation...	/
RequiresImpl	f1-implemente...	/
RequiresImpl	f2-implemente...	/
RequiresImpl	f3-implemente...	/
RequiresImpl	f2-requires-f3	/
MutualExclusiveI...	f1-excludes-f3	/
✓ SelectedFeatureI...	Select_Feature...	FeatureM...
✓ SelectedFeatureI...	Select_Implem...	Impleme...
✗ EliminatedFeatureI...	f2_eliminated	f2
✓ SelectedFeatureI...	f3_selected	f3
✓ SelectedFeatureI...	Select_c2	c2
✗ EliminatedFeatureI...	Eliminate_f1	f1

Fig. 3. The S2T2 Configurator showing an integrated SPL model during configuration.

their implementation, and relations between them in one model and (2) this model can be used in an interactive configuration tool.

V. Product Configuration

After converting the given SPL artefacts into one integrated model, we can use our tool `S2T2 Configurator` to perform an interactive configuration ❸.

Whenever a model is loaded, the Configurator internally transforms it into a formal representation, which is used by a reasoning engine to keep the configured model in an consistent state, to calculates consequences of the user's decisions and, on demand, and to provides visual explanations for such consequences [2].

Figure 3 shows the Configurator with a very simple model with just three features f_1 to f_3 (left-hand side) and two components c_1 and c_2 (right-hand side). Within the feature model, we have two dependencies (f_1 and f_3 are mutually exclusive; f_2 requires f_3). The features and components are connected via requires edges, which represent that features are implemented by certain components.

In the example, the user decided earlier that f_2 is eliminated and f_3 is selected (this is indicated by the red cross in front of f_2 and the green check mark in front of f_3). From these decisions and information in the model the tool derived that, f_1 has to be eliminated and c_2 has to be selected. In the screenshot, the user just asked why c_2 was selected (see the open context menu). The tool responded by highlighting f_3, c_2 and the requires edge between them.

We can apply this tool to handle more realistic models, which have been derived from a real Simulink implementation model (using the model transformation briefly explained earlier).

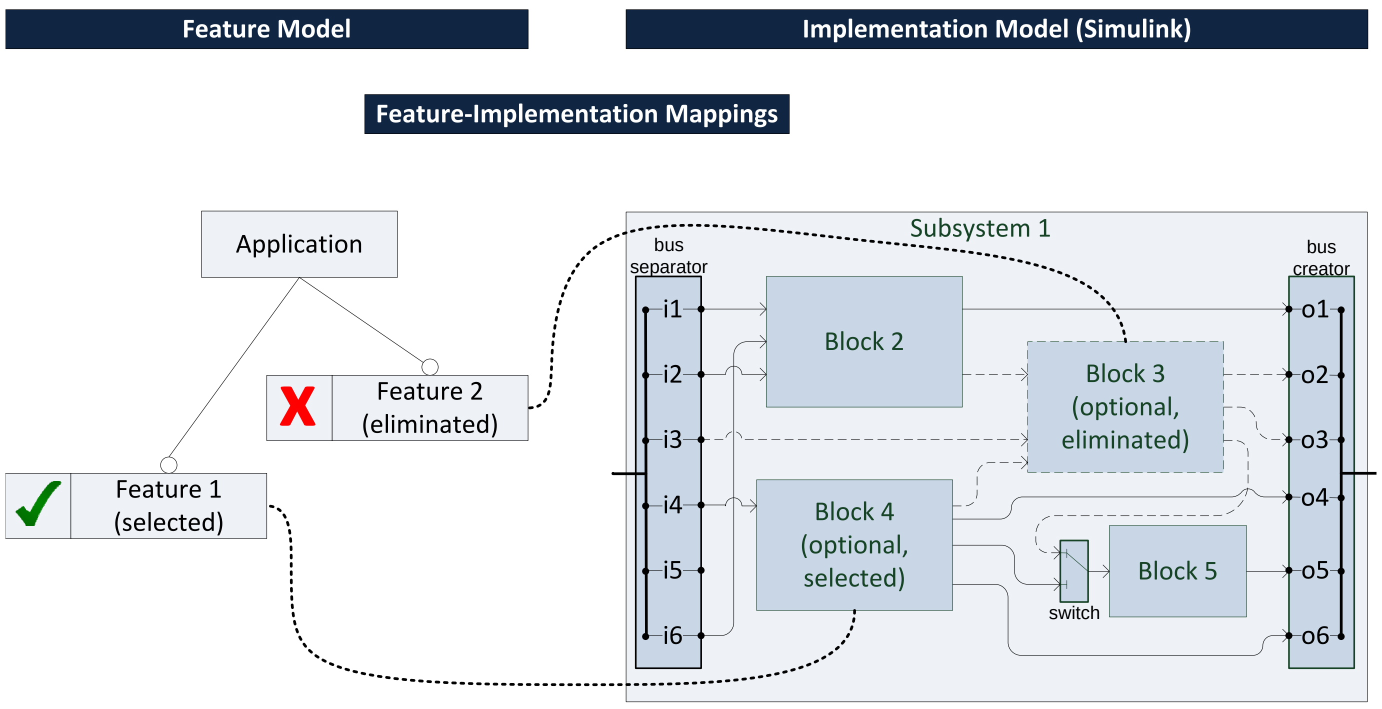


Fig. 4. Pruning of models.

VI. Product Derivation

Given the product configuration $\boxed{A_a}$ we now have to turn this into an executable product. In the overview in figure 2 this corresponds to the process of *Product Derivation* ❹.

A. Negative Variability

The first step towards the executable product is the derivation of the Application Simulink Model $\boxed{C_a}$.

Here, we apply a well-known technique called negative variability: The Domain Simulink Model $\boxed{C_d}$ contains the *union* of all possible product-specific Simulink Models $\boxed{C_a}$. Based on the configuration decisions in the product configuration $\boxed{A_a}$ we then copy the Simulink models while filtering out all elements, which correspond to eliminated features. Hence, the term negative variability.

This technique can, for instance, be implemented with ATL model transformations (as demonstrated in earlier work [10]) or with openArchitectureWare's XVar component. For the approach discussed here we are currently experimenting with a connector that connects our Configurator to openArchitectureWare [8].

B. Pruning

The technique of negative variability is a first step, but it is not sufficient to get a consistent model. We will briefly discuss two situations, where we have to adapt more than just removing some affected blocks.

The first situation arises with alternative features. With such alternative groups of features, often the outputs of the corresponding implementation blocks are connected to the same port of a third block. This pattern is not a legal Simulink model, because Simulink does not know anything about the alternative group and the elements which will be removed later to obtain a legal model.

Hence, to create and test such a model within the Simulink tools, we have to introduce helper mechanism like *Switch* blocks. When we later apply negative variability, these helper mechanisms have to be adapted (or removed) as well. An example is depicted in the figure 4, where two signals lead into `Block 5` and are handled by a switch block. Whenever, only one of these two signals is left, we can remove the switch block altogether.

A second situation, where we have to adopt additional components in the Simulink model are *Bus* elements. These elements allow to combine multiple signals into one logical bus, to simplify the model (*Bus Creator*). In a different location in the same model, such a bus can again be separated into the single signals (*Bus Separator*). Whenever we apply negative variability, some signals within these buses might have to be removed (because the blocks providing these signals are no longer present). Hence, we have to adapt the bus. See the *bus creator* and *bus separator* in figure 4. In the given example, the signals i_3, o_2 and o_3 could be pruned.

VII. Discussion

In this work we implemented an approach, where we combined the feature, the mapping, and the implementation model into one big feature model. To this end, we transformed

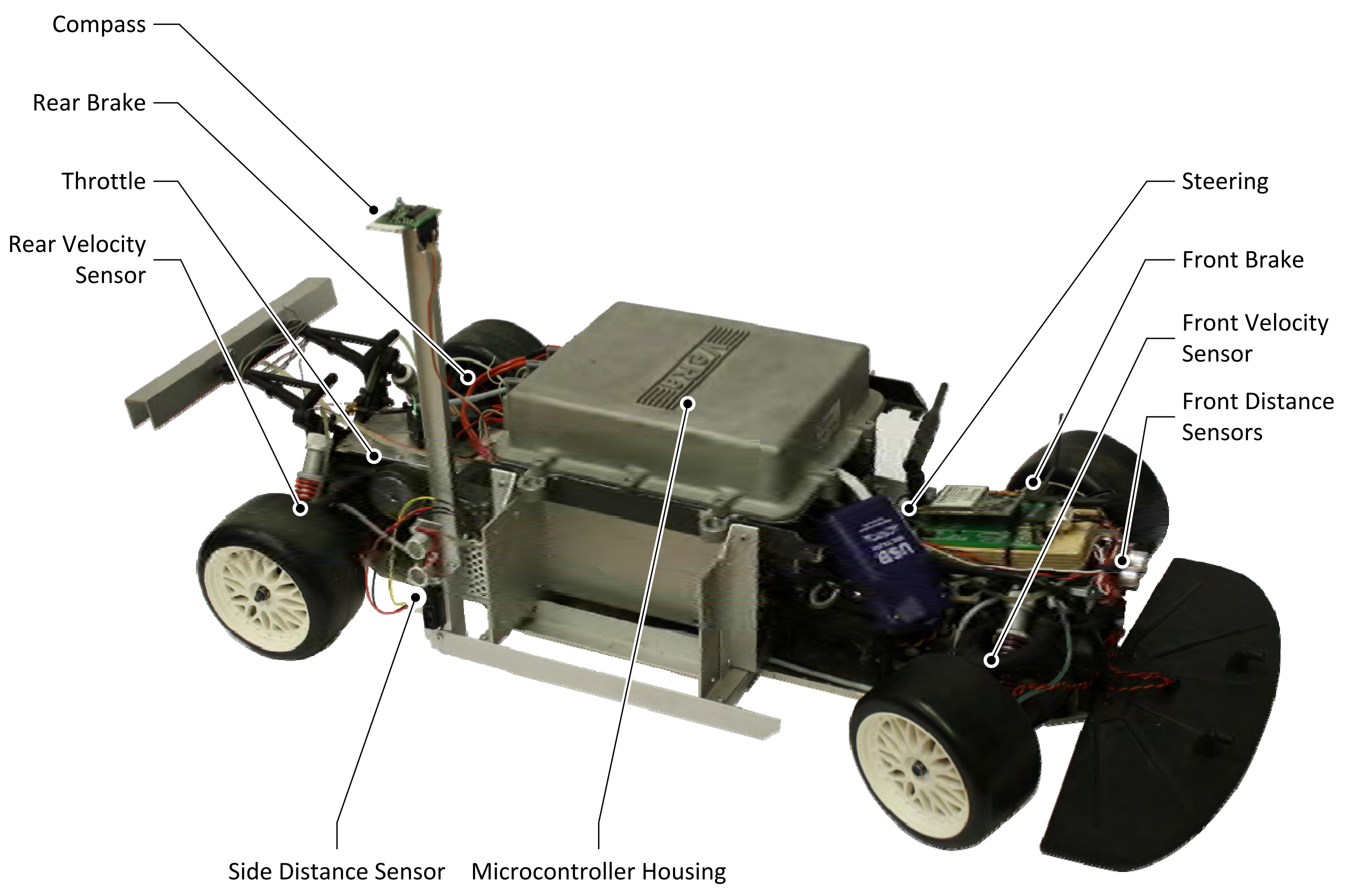


Fig. 5. The model car of the VeRa Rapid Control Prototyping platform.

the Simulink blocks of the implementation model into features and the mappings into constraints between the features and the blocks represented as features.

We experimented with this approach by using a product line of parking assistant applications, which is implemented using a Simulink model and a Rapid-Control-Prototyping system called VeRa. The model of the parking assistant can be simulate within Simulink or executed on a 1:5 model car, which is shown in figure 5. The application contains variability because of alternative distance sensors and an optional compass sensor, which helps the car to orientate itself in a parking bay.

This model contains a large number of blocks, subsystems and buses. Two parking algorithms are implemented to deal with the variability: One which uses the compass information and one without this information.

The transformation of a big Simulink model like our parking assistant into a feature model does not need remarkable time. So scalability in terms of execution time of the transformation seems to stay in reasonable bounds.

However, for larger models, during configuration the cognitive complexity and usability become an issue. Some techniques how to mitigate these problem with interaction techniques introduced to our `S2T2 Configurator` have been described in [3]. Up to now, we do not know if our approach scales in terms of usability. Hence, we intend to use large, realistic Simulink models and evaluate the configuration approach.

During the implementation we made the experience that Simulink is less strict about the syntax and the contents of values and parameters. This causes problems during transformation because the mechanisms further down the tool chain, such as Eclipse Modeling Framework (EMF, for handling models), ATLAS Transformation Language (ATL, for transforming models) and our Configurator are more strict about values.

For instance, it is perfectly legal to name a Block ”S-Function” in Simulink, actually including the quotes in the value. However, this will lead to technical problem when converting this to an EMF model. Similarly, Simulink does not care if names of blocks are unique. In EMF, however, it is desirable to have unique names since these are used as identifiers in references.

VIII. Related work

In earlier work we presented the basic architecture of the configurator [2] and discussed interaction techniques [3]. Here we extend this work by (1) a new approach of visualizing features and implementation, (2) using a configured feature model for product derivation and (3) pruning approaches to adapt the implementation model. The whole approach is evaluated using a parking assistant as application and a tool chain using a Rapid Control Prototyping System.

Approaches which are related to our work can be roughly grouped in two categories, (1) approaches to deal with variability in domain-specific languages and (2) approaches to model variability in model-based development with Matlab/Simulink.

Weiland [11] addresses the challenges of variability in Matlab/Simulink. He uses marked standard Simulink blocks like switches to represent the different choices. Hence, the Simulink model contains the whole variability, a variant is then chosen by setting the corresponding parameters and selecting a specific signal path.

Kubica [12] starts from a feature model modeled in *pure::variants*, where the developer has to choose the desired features. Subsequently, the corresponding Simulink model is build automatically from templates and fragment models stored in the configuration tool.

There are other approaches, which are dealing with domain-specific techniques as well. For instance, Voelter and Groher [13] describe how to use openArchitectureWare [8] for Software Product Line Engineering. They use aspect-oriented and model-driven methods to generate products. To evaluate their approach they discuss a product line of Smart Home applications.

When dealing with variability, a typical challenge is the mapping of features or variation points to their implementation. Czarnecki and Antkiewicz [14] used a template-based approach where visibility conditions for model elements are described in OCL. Heidenreich et al. [15] present FeatureMapper, a tool-supported approach which can map features to arbitrary EMF-based models [16].

IX. Conclusions

In this paper, we presented an approach to the configuration of product lines within an existing tool for feature configuration.

The necessary translation from the implementation model into feature models and the targeted feature modeling language, present some limits with respect to expressive power. We can only "translate" model structures that are related to configuration, such as selection/elimination or $x-requires-y$ dependencies. More domain-specific concepts, e.g., voltages or oscilloscopes cannot be represented in a feature model in a meaningful way.

On the other hand, this translation enables us to configure an *integrated* model of the whole product line within one configuration tool. In particular, it provide the functionality described in the introduction:

1) The user can browse dependencies between features and the related elements in other models.
2) After applying configuration decisions he/she can immediately see consequences in related models.
3) It is possible to apply configuration decisions in the implementation model first and derive implications for the feature model from that.

In future work, we intend to improve (1) the product derivation mechanisms, including the connector which links our Configurator to openArchitectureWare and (2) the model transformation that implements the pruning.

X. Acknowledgments

This work is partially supported by Science Foundation Ireland under grant no. 03/CE2/I303_1 to Lero – The Irish Software Engineering Research Centre, http://www.lero.ie/.

References

[1] D. Beuche, "Modeling and building software product lines with pure::variants," in *12th International Software Product Line Conference (SPLC 2008)*, Limerick, Ireland, September 2008. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SPLC.2008.53

[2] G. Botterweck, M. Janota, and D. Schneeweiss, "A design of a configurable feature model configurator," in *Proceedings of the 3rd International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 09)*, January 2009. [Online]. Available: http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf

[3] G. Botterweck, D. Schneeweiss, and A. Pleuss, "Interactive techniques to support the configuration of complex feature models," in *1st International Workshop on Model-Driven Product Line Engineering (MDPLE 2009), held in conjunction with ECMDA 2009*, Twente, The Netherlands, June 2009. [Online]. Available: https://feasiple.de/public/proceedings-mdple2009.pdf

[4] J. Lunze, *Regelungstechnik 1.* Springer, 2004.

[5] A. Polzer, S. Kowalewski, and G. Botterweck, "Applying software product line techniques in model-based embedded systems engineering," in *6th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2009), Workshop at the 31st International Conference on Software Engineering (ICSE 2009)*, Vancouver, Canada, May 2009. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MOMPES.2009.5069132

[6] A. Polzer, G. Botterweck, and S. Kowalewski, "Variabilität im modellbasierten Engineering von eingebetteten Systemen," in *7. Workshop Automotive Software Engineering*, 2009.

[7] Eclipse-Foundation, "Eclipse Modeling Project." [Online]. Available: http://www.eclipse.org/modeling/

[8] "openArchitectureWare homepage." [Online]. Available: http://www.openarchitectureware.org/

[9] Eclipse-Foundation, "ATL (ATLAS Transformation Language)." [Online]. Available: http://www.eclipse.org/m2m/atl/

[10] G. Botterweck, L. O'Brien, and S. Thiel, "Model-driven derivation of product architectures," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE 2007)*, Atlanta, GA, USA, 2007, pp. 469–472. [Online]. Available: http://dx.doi.org/10.1145/1321631.1321711

[11] J. Weiland and E. Richter, "Konfigurationsmanagement variantenreicher simulink-modelle," in *Informatik 2005 - Informatik LIVE!, Band 2*. Koellen Druck+Verlag GmbH, Bonn, September 2005.

[12] S. Kubica, "Variantenmanagement modellbasierter funktionssoftware mit software-produktlinien," Ph.D. dissertation, Universität Erlangen-Nürnberg, Institut für Informatik, 2007, arbeitsberichte des Instituts für Informatik, Friedrich-Alexander-Universität Erlangen Nürnberg.

[13] M. Voelter and I. Groher, "Product line implementation using aspect-oriented and model-driven software development," in *11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, September 2007. [Online]. Available: http://www.voelter.de/data/pub/VoelterGroher_SPLEwithAOandMDD.pdf

[14] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *GPCE'05*, Tallinn, Estonia, September 29 - October 1 2005. [Online]. Available: http://dx.doi.org/10.1007/11561347_28

[15] F. Heidenreich, J. Kopcsek, and C. Wende, "Featuremapper: Mapping features to models," in *ICSE Companion '08: Companion of the 13th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 943–944. [Online]. Available: http://doi.acm.org/10.1145/1370175.1370199

[16] Eclipse-Foundation, "EMF - Eclipse Modelling Framework." [Online]. Available: http://www.eclipse.org/modeling/emf/

Integrated Modeling of Software Product Lines with Feature Models and Classification Trees

Sebastian Oster
Real-Time Systems Group
Technische Universität Darmstadt

Florian Markert
Computer Systems Group
Technische Universität Darmstadt

Andy Schürr
Real-Time Systems Group
Technische Universität Darmstadt

Werner Müller
Global Systems Engineering Methods
Adam Opel GmbH

Abstract—**Software Product Lines (SPLs) are an approach to improve reusability of software in a large number of products that share a common set of features. In SPLs, Feature Models (FMs) are frequently used to model commonalities and variabilities. However, according to the best of our knowledge, there are no approaches to automatically generate test cases on the basis of a stand-alone FM. We introduce a method, which fills this gap. In single system software testing, Classification Trees (CTs) are a proven approach for generating test cases derived from the original system specification. In this paper, we explore the relations and similarities between FMs and CTs and integrate both methods to a unified approach called *Feature Model for Testing* (FMT).**

I. Introduction

SPL-engineering is one of the most promising approaches of the Software Engineering community to reduce the development costs, for as well as to increase the quality of families of similar software product instances [4]. As a consequence, software product line engineering is successfully used in various domains, including the domain of automotive software development. In this area, the combination of highly parametrized software of electronic control units (ECUs), together with an abundance of configuration options of networks of ECUs will soon lead to a situation, where (1) a single ECU may be instantiated in at least 10.000 different ways and (2) the software of a network of more than 50 ECUs in a single car may exist in millions of different configurations.

As a matter of fact, we are, therefore, running into a situation where each instance of a certain brand of car possesses a unique configuration of the embedded software of all its ECUs. Testing all these millions of instances of an automotive SPL in the following traditional way is no longer feasible: create all actually used instances of an SPL and then develop for each instance a separate suite of integration test cases. Hence, the automotive industry as well as engineers from other domains are urgently looking for new methods how to systematically generate sets of software product instances that represent equivalence classes of instances with a sufficiently similar behavior from a system integration testing point of view. Furthermore, model-based and more traditional black and white box testing approaches are adapted in such a way that families of test models and derived test cases can be developed, together with semi-automatic procedures that allow one to select the appropriate test cases for a specific SPL instance.

Examining more closely the state-of-the-art of SPL development and software testing approaches in the automotive industry we see that various kinds of feature modeling concepts and tools are used for the design of SPLs and the selection of needed instances [4], [21]. On the other hand, CTs and related tools such as CTE [1] are successfully used for black-box testing of single product instances. We are not aware of any proposal how to combine feature modeling concepts used for the description and selection of features (parameters, options, ...) of SPLs with CT-concepts used for the description and selection of test case parameters of selected product instances—despite of the fact that the borderlines between feature selection at compile time and input parameter selection at runtime are blurred, and the same parameter may either be instantiated at compile time or flash time, to unlock a specific function or changed at runtime to activate a certain mode of operation.

To overcome these problems we will first present in Section II of this paper the fundamentals of SPL description by means of FMs and the specification of parameter equivalence classes for testing purposes by means of CTs. Furthermore, this section introduces our paper's running example, a subset of a case study provided by the Adam Opel GmbH which is a subsidiary of GM (General Motors) Corp. Afterwards, we explain in more detail the state-of-the-art of systematic testing approaches of SPLs and black box testing with CTs in a related work section. Section IV then systematically compares the basic modeling elements of FMs, similar to the developed FMs in FODA [14] and the classification approach as supported by the tool CTE [1]. Based on this comparison, a tight integration of both modeling languages is proposed and the abstract syntax of the resulting feature and test parameter modeling language "FMT" is presented. In Section V we describe the derivation of a product with corresponding test cases from the FMT. Additionally, we discuss an approach of testing SPLs using FMT. The conclusion summarizes the advantages of such an integrated feature and parameter equivalence class modeling approach and lists a number of ongoing and future research activities.

II. Fundamentals

The contribution of this paper is to generate variants and corresponding test cases on the basis of one representation of the SPL.

A. Running Example

Our running example is a very simple subset of hardware and software components of the recently released Opel Insignia. We restrain ourselves to four sensors, two actors (engines), and one software component. The sensors are *rain light sensor (RLS)*, *turn indicator sensor (TIS)*, *steering angle sensor (SAS)*, and the *vehicle speed sensor (VSS)*. The *RLS* detects rain and the *TIS* indicates the driver's choice to turn left or right. *SAS* senses the steering angle and the *VSS* senses the speed of the car. Two different types of engines serve as hardware features: a *1.6 liter* and a *2.0 liter turbo* engine. Additionally, we use one feature of the (Adaptive Forward Lighting) AFL+ technology. The *bending light* is a functionality which belongs to AFL+ and realizes an adaptive curve light. All parameters presented in this paper are provided by the Adam Opel GmbH. We use the running example to exemplify the differences and commonalties between FMs and CTs and to motivate our approach of integrating both to a Feature Model for Testing (FMT).

B. Product Lines and Feature Models

SPLs provide a high level reuse of software in a specific problem domain [4], [21]. FMs are frequently used to describe an explicit representation of the commonalities and variabilities in an SPL. FMs consist of features each representing "a system property that is relevant to some stakeholder" as defined in [6]. One major advantage of using FMs to model SPLs is that they offer a very intuitive way to represent commonalities and variabilities. However, FMs by themselves are insufficient for a complete modeling of an SPL. Usually FMs are complemented by development artifacts such as UML diagrams or code fragments that are traced to the corresponding features. An FM provides a tree-like structure and incorporates different node notations and cross-tree-constraints. The first feature model was introduced by Kang et al. in [14] as part of FODA, in 1990. In the FM of FODA node notations like, mandatory, optional, and alternative features can be modeled. It is also possible to use require and exclude constraints between features, which are described textually. Since the introduction of FODA, further extensions of FMs were introduced to improve precision and expressiveness, including amongst others cardinalities, probabilities, and weighting. We can employ cardinalities to formulate the different notations of features and groups of features [7]. The probabilities can be based on empirical data and/or system specifications and are used to state that a certain feature is more likely than another one [8]. Weights can be used to represent cost factors of features to help the engineers to build products appropriate for a certain budget [13]. Czarnecki et al. summarize some existing extensions of the FODA FM [6]. However, our FM is very similar to the original FODA with additional cardinality extension [7]. Table I depicts the used notation. We do not want to discuss

Graphical Notation	Cardinality	Formal description
Single features		
edge with filled circle	1..1	feature is mandatory
edge with unfilled circle	0..1	feature is optional
Group of features		
filled circles and connected edges	1..n	choose exactly one feature
unfilled circles and connected edges	n..m	choose any combination of features, but at least one

TABLE I
NOTATION

every FODA extension because our approach is not limited to a certain FM. We also aim e.g. to support the Orthogonal Variability Model (OVM) proposed by Pohl et al. [21]. A detailed description of the relation between FMs and OVMs is given in [18]. We, therefore, assume that our unified approach using FMs may also use the OVM approach.
The FM in Fig. 1 shows an excerpt of the Opel Insignia FM. All dependencies and constraints within the FM have to be taken into account when deriving a product.

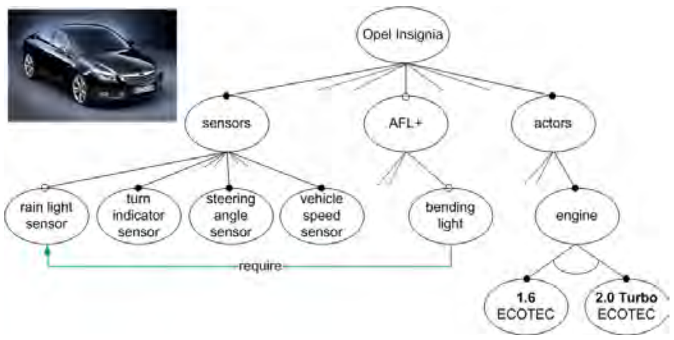


Fig. 1. Feature Model of Opel Insignia

C. Software Testing with Classification Trees

After extracting a variant from the FM, suitable test cases need to be built. For this purpose the CT-method was introduced by Grochtmann and Grimm [11]. It provides an approach to black box testing. Test cases can be extracted by defining rules for valid input combinations. A CT consists of *classifications* (boxes with bold lines), *compositions* (boxes with thin lines), and *equivalence classes* (values in brackets). Each *equivalence class* represents a disjoint subset of parameter values for a *classification*, while the *composition* splits complex input parameters into a number of subcomponents. To generate test cases from a specification using CTs the following procedure needs to be applied:

1) evaluate the specification and identify all *classifications* with the corresponding *equivalence classes*
2) build the CT
3) fill the test case table with respect to the CT using parameter value combination heuristics

4) extract all possible test cases from the test case table by identifying valid combinations of *equivalence classes* omitting illegitimate samples out of the test set

Fig. 2 depicts a variant of the Insignia SPL and a corresponding CT to test this instance. *Sensors* is a *composition*, while *turn indicator* is a *classification* of the *equivalence classes* representing test values. The model is extracted from the specification of the product instance. Test cases can be derived easily by choosing a valid combination of *equivalence classes*. This model can e.g. be used to check the output characteristics of the AFL+ dependent upon the sensor intervals. The range of the intervals is defined by the designer. It depends on suitable test scenarios chosen by the designer or extracted from the specification. A test case table is built, which incorporates all relevant test cases in an abstract way.

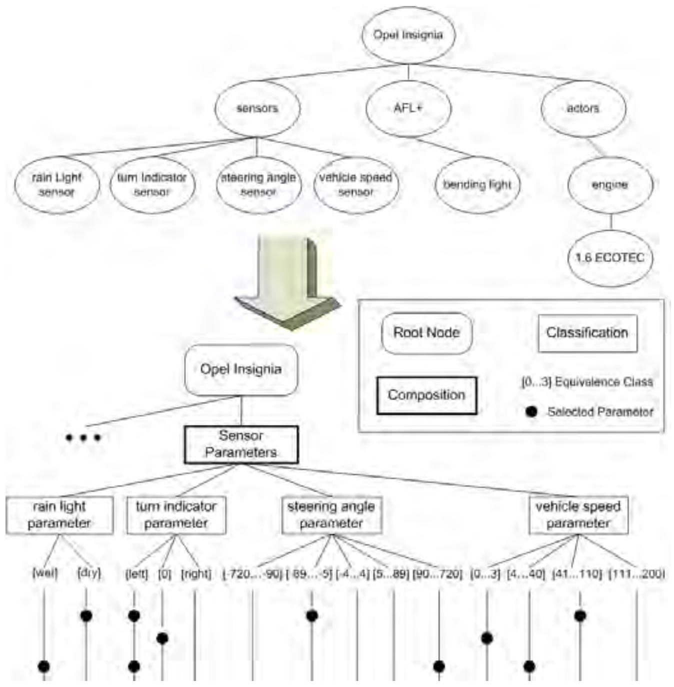


Fig. 2. Classification tree of selected product instance

In Fig. 2 three test cases for the CT of the running example are given. Each test case consists of the *equivalence classes* marked by the black dots in a horizontal line.

III. Related Work

In this section we focus on SPL-testing and the role FMs play in that context. We also present research activities related to software testing using the CT-method to generate test cases.

A. Testing SPLs

Miscellaneous approaches exist dedicated to SPL testing. Although there are no real FM-based testing approaches, many test methods partially use the FM of the SPL under test. A summary of methods is given in [28]. The authors distinguish between the following approaches to integrate testing into the development process of SPLs:

Product-by-product testing: In the majority of cases, each instance of an SPL is tested individually. This method is called product-by-product testing. Each product instance may be tested using the CT approach. However, this method is very exhaustive and normally not practicable. For instance, in the automotive field a single ECU like the engine control unit may be instantiated in at least several 10.000 different ways. Since a car may consist of up to 50 ECUs this results in millions of different configurations. An individual test of all configurations is feasible neither at the end of the assembly line nor during the development process. One method to improve product-by-product testing is to identify a minimal set of products which is representative for all other products. However, finding a minimal test set is an NP-complete problem [26]. Different heuristics are used to approximate a minimal test set. A very promising procedure is mentioned in [26]. The author uses a simplified version of the OVM approach. Members for the representative set of products are selected on the basis of requirements. That means that certain products are chosen so that all requirements are verified at least once. Optimization problems are formulated to produce a minimal test set.

In [20], an approach with a motivation similar to [26] was published. It uses dependencies derived from architecture and implementation to extend the FM of the SPL under test. Subsequently, pairwise testing that considers these dependencies is used to generate a representative set of products.

In both approaches the generation of test cases with appropriate input parameter values is out of scope.

Incremental Testing: In this approach, the first product derived from the SPL is tested individually, for instance by using the CT-methodology. With respect to the commonalities between the different products, the following products are tested using regression testing techniques [17]. The challenging part of this approach is to identify those parts of a product which stay unchanged and those which vary. The question arises if only the modified and added parts have to be tested. In addition, one has to find out if the modified parts can be tested individually or if some of them have to be tested in combination with unchanged components.

Reusable assets: The goal is to create reusable test assets. To ensure reusability, these assets are created during domain engineering. For each product these assets are customized during application engineering. Pohl et al. apply model based testing techniques. The authors use activity diagrams which are developed in domain engineering, based on requirements, and customized during application engineering to derive test cases. The so called ScenTED approach focuses on the reuse of test cases [25], [24], [22]. It uses substitution in order to derive configuration specific test cases. This is a very promising approach which we will take into account in our future work. However, test parametrization and the selection of proper values is still an open problem.

Division of responsibility: In [28], this method is described according to the levels of the development process, for instance the V-model. For example, unit testing can be executed during

domain engineering and the other levels of the V-model could be executed in the application engineering phase.
All approaches confirm that testing is very challenging in SPL-engineering due to the high level of variability. However, the fact that the parametrization within an SPL leads to an additional degree of variability is often neglected as well as the fact that the borderlines between parameters that model variability at design time and parameters that are instantiated at runtime are often moving.
Furthermore, we are not aware of any SPL testing approach that combines SPL variant selection strategies with parameter value selection strategies as supported by CT. The tool pure::variants [23] offers e.g. the option to store parameter information in attributes of features, but gives no support for the definition of equivalence classes etc. The approach published in [19] is as far as we know the most similar SPL testing method to FMT. It uses one decision model to represent the variability of an SPL as well as to document input parameters of the regarded system. An integrated SPL variant and parameter value selection process is presented with rather promising evaluation results. Compared to FMT presented here the decision model is considerably less expressive and the introduced selection process is considerably less expressive than the algorithms for SPLs and the heuristics developed for CT-based black box testing purposes.

B. Black-Box Testing Products with CT

CTs are a widely used technique for test case generation. There are many publications related to this topic. A CT can be used to test a single product instance derived from the FM. The resulting product instance has specific actors and sensors, which interact complying to the specification. A CT splits the input domain of the sensors into relevant *equivalence classes*. These classes can be used to test an actor that is part of the product instance. Several approaches to improve and extend CTs like the Classification Hierarchy Table [3], the Classification Tree Transformer [27], Class Graphs [15], adding attributes [16], and introducing a time line [5] have been discussed. There are tools like the Classification Tree Editor for Embedded Systems (CTE/ES) that support the designer when trying to build a CT and its test table. The CTE/ES provides a graphical user interface that enables the designer to build a CT and the corresponding combination table.
All CT-based testing approaches we are aware of share the drawback that they deal with single product instances only and thus are only compatible with a product-by-product SPL testing approach.

IV. Unified Approach - FMT

As introduced in the preceding sections FMs and CTs have been used in software engineering for rather different purposes until now. An in-depth comparison of the modeling language constructs of FMs and CTs in the sequel reveals that both modeling approaches share a majority of their concepts. Therefore, this section is structured as follows: the first subsection starts with the in-depth comparison of FM and CT modeling constructs. Based on the results of this comparison, the following subsection then selects a minimal number of language constructs that correspond to a superset of the concepts of both FM and CT. Finally, the last subsection introduces a metamodel that captures the essential design of the new integrated FMT modeling language from an abstract syntax point of view.

A. FMT Language Constructs

In this section we develop language constructs which present a minimal list of abstract language concepts that is a superset of the concepts of FM and CT. Table II lists the constructs of FMs and CTs and the corresponding constructs for the unified approach: FMT. First, we have to define which kinds of nodes the FMT needs to support. CTs distinguish between *compositions*, *classification*, and *equivalence classes*. *Composition* and *classification* (line 1 in Table II) in CTs are nodes with child elements and *equivalence classes* (line 3 in Table II) are leafs in a CT. In FMs only *compound features* contain child elements and leafs are always *features*. To integrate CTs and FMs with regard to the different kinds of nodes, we need to examine the differences. *Compositions* are always mandatory and *classifications* are always optional nodes. As a consequence, we can use the *compound feature* known from FMs, and use cardinalities to state that the *compound feature* is either optional (0..1) or mandatory (1..1). Therefore, we adopt the *compound feature* for the FMT approach. The leafs of CTs and FMs differ obviously. On the one hand, an *equivalence class* represents a value or range of values of a parameter necessary for testing. On the other hand a *feature* in FMs can be any feature or property of an SPL. To realize an integration we need a leaf, which is capable to represent both information: a *feature* in general and a representation of values of parameters. Another difference, which we have to take into account for the integration, are the

	Feature Model	Classification Tree	FMT
1.	compound feature	composition, classification	compound feature
2.	Feature	none	Feature
3.	none	equivalence class	atomic feature
4.	mandatory feature (1..1)	composition	mandatory feature (1..1)
5.	optional feature (0..1)	classification	optional feature (0..1)
6.	(1..n) features	equivalence class	(1..n) features
7.	(n..m) features	none	(n..m) features
8.	cross-tree-constraints	only between equivalence classes	cross-tree-constraints
9.	feature attributes	none	feature attributes
10.	feature types	none	feature types

TABLE II
LANGUAGE CONSTRUCTS

cardinalities. In FMs *features* and *compound features* may have four different types of cardinalities to describe commonalities and variabilities. In CTs three of them are present: *composition* (1..1), *classification* (0..1), and *equivalence classes* (1..n). We adopt the missing cardinality of FM to ensure that the FMT is as expressive as the original FM. Furthermore, all four types of cardinalities may be used on all levels of an FMT tree in contrast to the much more restrictive approach of CT.

In addition, we allow cross-tree-constraints between all nodes of the FMT (line 8 in Table II). Finally, we adopt the node attributes and node types from feature modeling (line 9 and 10 in Table II).

B. Concept

We now describe the integration of FM and CT by means of a metamodel depicted in Fig. 3. We developed this class diagram on the basis of Table II. Please note that the depicted class diagram is a small extract of the complete FMT metamodel that illustrates the concept of the unified approach. It does e.g. not contain any information concerning the test case generation. The characteristics of the nodes of the FMT approach are described using the following classes: `Compound Feature`, `Feature`, and `Atomic Feature`. The last one can either be a feature representing its property in form of a literal or an interval. This is realized using the two classes `Literal` and `Interval`. These classes may represent:

- a value or a range of values of test parameters as known from equivalence classes in CTs
- a value or a range of values of parameters needed for product instantiation purposes
- the labels of features known from FMs.

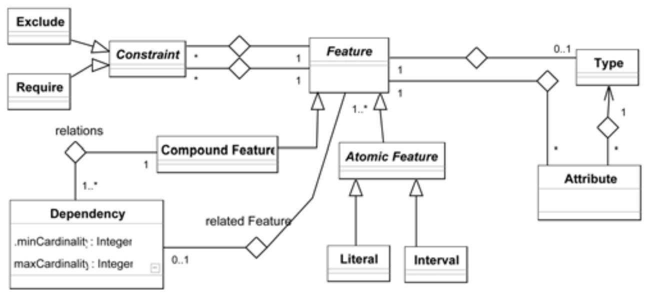


Fig. 3. Simplified metamodel of the unified approach

A node in the FMT can only be a `Compound Feature` or a leaf node (`Literal` or `Interval`), because the classes `Feature` and `Atomic Feature` are abstract. `Compound Feature`, `Literal`, and `Interval` inherit properties from `Feature`. All nodes in the FMT may have a `Type` and an arbitrary number of `Attributes`. In a `Type` the information of the data type of a feature is stored if existent. This is important, for instance, to distinguish between parameters of an integer or real data type. `Attributes` can store any information of the node. To obtain sufficient information to properly plan the test effort it is for example important for vehicle OEMs to embed information about realizing a feature in hardware or software. Additionally, we can apply constraints between `Features` and, therefore, also between `Compound Features` and `Atomic Features`. According to FMs we allow `Require` and `Exclude` constraints between all nodes. Since we want to adopt the cardinalities, describing the relation of features or groups of features to their parent node, we model a relation between `Features` and `Compound Features` by means of the class `Dependency`, which contains the cardinality constraints as attributes (minimum and maximum cardinality).

Fig. 4 depicts an object diagram that shows how dependencies and cardinalities are used to distinguish between the four different categories of subfeatures of Table II. A `Feature` is

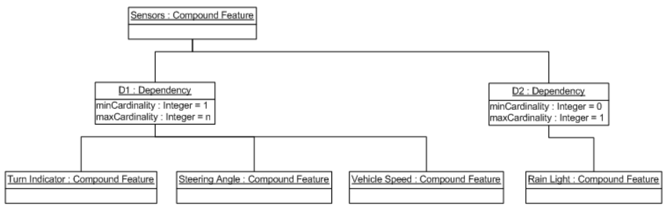

Sensors : Compound Feature
D1 : Dependency
minCardinality : Integer = 1
maxCardinality : Integer = n
D2 : Dependency
minCardinality : Integer = 0
maxCardinality : Integer = 1
Turn Indicator : Compound Feature
Steering Angle : Compound Feature
Vehicle Speed : Compound Feature
Rain Light : Compound Feature

Fig. 4. Object Diagram describing cardinality in FMT

always connected to its `Compound Feature` by a dependency. Therefore, `Dependencies` with different cardinalities can be placed beneath a `Compound Feature`.

V. Application

In this section we briefly describe the application of our FMT approach by generating test cases for our running example. Additionally, we discuss a tool, which is under development in our research groups.

A. Generating a test case

We demonstrate the ability to derive products and test cases on the basis of the FMT using our running example depicted in Fig. 5. We now derive two different products and present the handling of the parameters. The two products differ because they use different types of engines which results in different configurations. The 2.0 Turbo ECOTEC engine allows a vehicle maximum speed of 240km/h and, therefore, needs to be tested above 200km/h. The 1.6 ECOTEC engine is limited to 192km/h and does not require the test instance for vehicle speeds above 200km/h. Therefore, when testing the product containing the 1.6 ECOTEC engine, the *equivalence class* representing a speed range between 201 and 250 km/h has to be disregarded.

For both instances we derived some example test cases, which was done according to the well-known procedure described in section II-C. In the following we will describe the FMT in Fig. 5 which integrates the information of the FM of Fig. 1 and the CT of Fig. 2. Leaf nodes of the FMT, therefore, either represent basic features of the Opel Insignia SPL or *equivalence classes* of (runtime) parameter values. All non-leaf nodes of the FMT are inherited from the FM of Fig. 1, whereas leaf nodes are inherited from Fig. 1 and 2. The nodes of the new FMT have to be interpreted as follows:

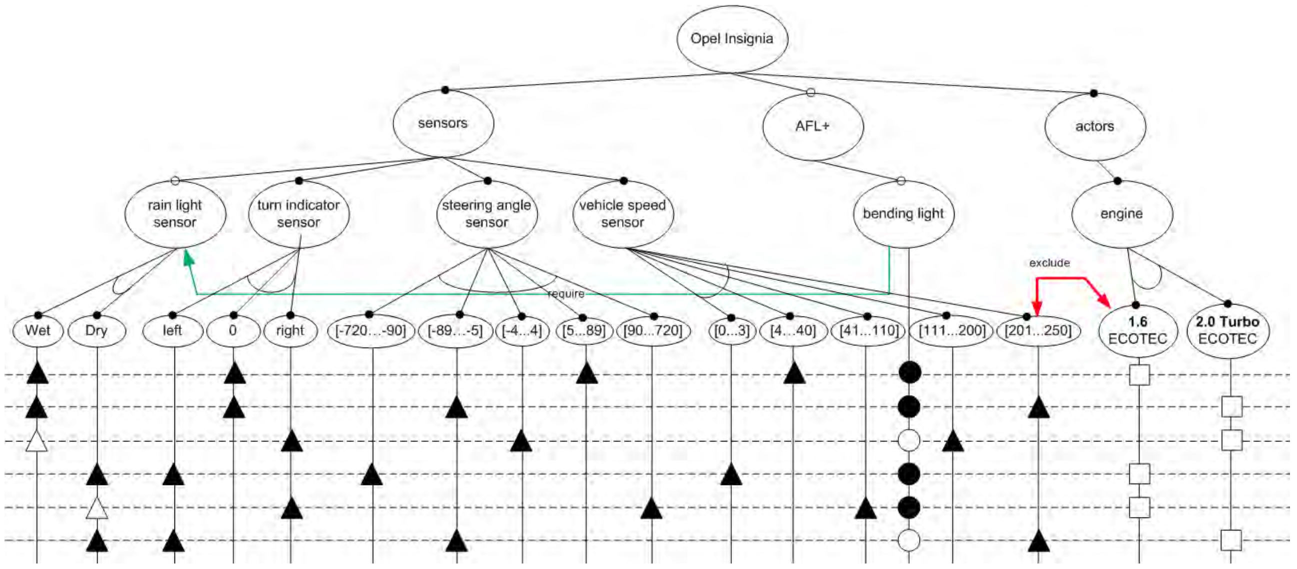


Fig. 5. FMT of the running example

- All nodes below the *sensors* node represent optional or mandatory Opel Insignia sensors together with their output parameter value definitions which are used as input parameters for control function test cases.
- All nodes below the *actors* node represent available actors (actuators) for the Opel Insignia such as different types of engines. Input parameters that control the behavior of these actors have been omitted due to lack of space. These missing parameters are output parameters of to be tested control functions. Specifications of their values can be used as oracles during the execution of test cases.
- A node like *bending light* represents a group of control functions that shall be tested. Again due to lack of space we do assume that *bending light* consists of a single function only which consumes output values of a subset of all sensors and produces input values for a subset of all actors defined in Fig. 6.

Node attributes (which are not visualized in Fig. 5) are used to distinguish these different categories of nodes of FMTs as well as for other purposes like the definition of additional node selection constraints (cf. metamodel of Fig. 3). Furthermore, Fig. 5 shows that the optional *bending light* functionality still requires the optional rain light sensor (as well as all other mandatory sensors of our SPL). The fact that the *bending light* control function also requires input values from the three other mandatory sensors is not visualized in Fig. 5. A more realistic FMT splits the *bending light* functionality into a number of subfunctionalities such as curve light, rain light, city light, or highway light which have to be tested separately. As a consequence we have to distinguish between features (functions, parameters, sensors, actors) that are part of a regarded product instance, but irrelevant for a specific test case, and features that are directly involved in a specific test case. The test case specifications in the lower part of Fig. 5 use black and white shapes for these two different purposes.

The selection of a specific variant and associated test cases is specified in a style adopted from CTs (due to lack of space the FMT metamodel of Fig. 3 does not cover these elements). Different shapes on vertical lines are used to distinguish between the following three cases, when a certain feature or parameter is selected:

1) square: selection at design time
2) circle: selection at installation time (flash time)
3) triangle: selection at runtime

The first two options correspond to the selection of a certain variant in an FM, whereas the third option corresponds to the selection of test cases with parameter values in a CT.

Regarding the bottom part of Fig. 5 we can see that the type of engine is of course selected at design time. The *bending light* functionality can be added or removed by firmware updates as long as the optional rain light sensor is present. The fact that all presented variants with their associated test cases do contain the optional rain light sensor is implicitly represented by the fact that all test cases possess a parameter value definition for this type of sensor (Wet or Dry). Black triangles represent Wet and Dry values that are actually used in a specific test case, whereas white triangles represent the fact that the *rain light sensor* is present and computes output values that are not needed as input for the just regarded test cases. Finally, Fig. 5 shows that four of the six depicted test cases are related to the functionality of the *bending light* (black bending light circles). In general, the distinction between black and white shapes related to other nodes of the FMT reflect the information which features and parameter values are relevant for which test case. It consists of nodes describing the product line and nodes for the test cases. Test case values are depicted in brackets.

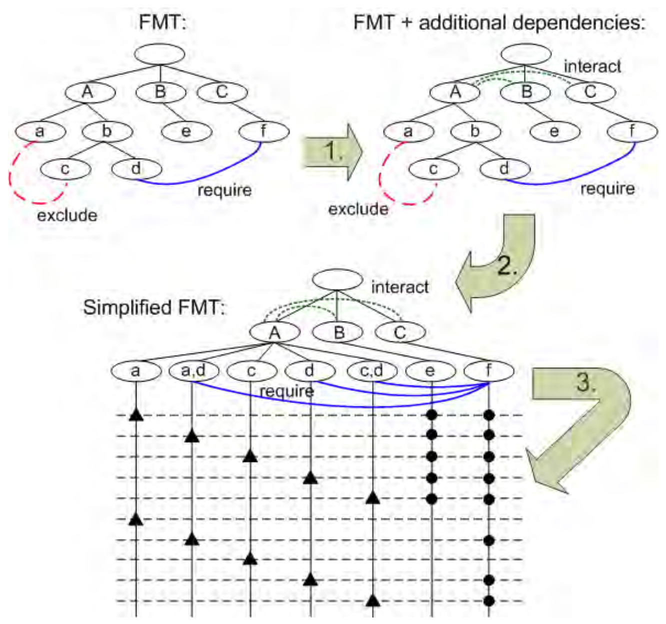


Fig. 6. Using FMT for SPL testing

B. Testing SPLs using FMT

In this subsection we describe in more detail how the FMT approach is used for SPL testing purposes, i.e. we sketch a methodology how to generate the bottom part of Fig. 5 automatically. For this purpose we describe the current state of development of our MoSo-PoLiTe project, which realizes the test methodology described in [20], but use FMT instead of FM. The generation of a representative set of products is subdivided into three steps [20].

1) Adding dependencies derived from system architecture and code as additional edges in the FMT.
2) Simplification of the FMT such that the resulting FMT uses a minimal set of modeling concepts.
3) Integrated selection of variants and runtime parameter values using the pairwise combination approach of [20]

Fig. 6 depicts the individual steps. We refer to [20] for further details. To use the FMT approach we are currently developing an FMT tool suite using MOFLON and GEF [10]. MOFLON is a meta-CASE tool for rapid development of CASE tools and tool adapters [2] developed at the Technische Universität Darmstadt. MOFLON supports model analysis, model transformation, and model integration for standard modeling languages like UML or domain-specific modeling languages. The abstract syntax of the new FMT modeling language as well as its static semantics rules are described using the OMG metamodeling approach supported by MOFLON, i.e. a combination of MOF 2.0 and OCL 2.0. Using this description as input we can generate an FMT model repository implementation together with all specified static semantics rules in Java. Furthermore, a generic text-oriented user interface for the definition of FMT instances is generated, too. The implementation of the user interface of a visual FMT editor on top of GEF is ongoing work. Model transformation rules (graph transformations) can be used to implement automatically executable FMT transformations. The last processing step of Fig. 6 has been implemented by modifying an existing Java implementation of a pairwise testing tool. The modified implementation in addition takes all kinds of dependencies between FMT nodes into account. The incorporation of CT-parameter value selection heuristics dealing e.g. with illegal or stress parameter *equivalence classes* (cf. Section II-C) is subject of ongoing research activities. The same is true for the first processing step depicted in Fig. 6. Right now dependencies that capture the fact that certain features or parameters interact from an implementation point of view have to be added manually. The adaption of ideas how to automatically derive this information from SPL architectures or code is also subject to future research activities.
According to the approach described in [20] we use the pairwise combination method to generate a representative set of products. At this point the major advantages of the FMT approach comes into play. We can generate the test cases for the selected products semi-automatically as described in the previous section. To summarize, we benefit from several advantages using FMT instead of FM. The FMT is more precise than FM when it comes to parametrization and for each product of the representative set we can derive the corresponding test cases semi-automatically. Likewise, the FMT describes parameters explicitly, therefore, we can consider dependencies which only occur between certain values of parameters.

VI. Conclusion and Future Work

Within this paper we have presented a new approach how to integrate SPL engineering with feature models (FMs) and black-box testing of system functions with CTs. Our motivation for this line of research is based on the fact that both FM- and CT-based methods are, e.g., well established in the automotive industry for embedded software system development purposes. On the one hand FMs are a suitable modeling method to describe commonalities and variabilities of an SPL. CTs, on the other hand, support the generation of test cases, using equivalence classes of parameter values of a regarded system function. We are not aware of any integrated approach that combines both methods for the generation of variants as test candidates and the associated test cases. Today, FM-based methods are first used to select one variant after the other; then for each of these variants a separate CT has to be defined which is then used to guide the related test case selection process. The integration of FM and CT in the form of the presented new FMT (Feature Model for Testing) method seems to be the perfect symbiosis of two very promising and widely used techniques. The key advantages are: (1) we use a single model for SPLs and test case generation, (2) approaches known from FMs and CTs can still be applied, and (3) generation algorithms of variants and test cases can be combined. We are currently working on different projects using FMT for SPL testing purposes including the BMBF project feasiPLe. Ongoing and future research activities will address the following problems:

- Checking the consistency of the FMT with respect to an additionally available specification of the behavior of the studied system. The FMT needs to incorporate the complete functionality described in the specification. This must be done before generating test cases.
- In embedded systems, our main application area, real-time constraints play an important role. Furthermore, test cases often have to be executed in a specified order and continuous parameter values reflecting physical properties of a controlled system and its environment have to be synthesized from sequences of selected discrete parameter values using well-known interpolation methods.
- Defining a measure of completeness for the generated test scenarios with respect to an additionally available system behavior specification is challenging, too [26]. Completeness checkers are useful to evaluate the generated test cases and to find gaps and corner cases.
- The nodes of the FMT have to be extended to describe the cost of creating configurations and requirements priority which is essential for complex SPLs in the automotive sector.
- We are currently applying our approach to several SPL scenarios. These are real world examples and FMTs generated randomly.
- We apply the pairwise testing approach [20] to generate a representative set of test cases and measure the coverage using appropriate and well known coverage metrics.

We focus on model checking techniques to address some of the problems stated above. Hence, another field of our research is the use of model checkers for test case generation and FMT validation. For this purpose, we need to develop a tool that is able to translate the FMT and the resulting test cases into boolean formulas, which can be evaluated by a model checker. Methodologies to translate an FM into a boolean formula have already been introduced in [12], [9].
Finally, we have to address the problem that realistic complete FMT models cannot be displayed directly as depicted in this paper. Various methods have to be developed how to collapse/hide irrelevant substructures efficiently.

Acknowledgment

This work was partially funded by the *feasiPLe* project, "Feature-driven, aspect-oriented and model-driven Software Product Line Development", Federal Ministry of Education and Research (BMBF), Germany.

References

[1] S. Alekseev, R. Tiede, and P. Tollkühn, "Systematic approach for using the classification tree method for testing complex software-systems," in *SE'07: Proceedings of the 25th conference on IASTED*. Anaheim, CA, USA: ACTA Press, 2007, pp. 261–266.

[2] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr, "Adapting FUJABA for Building a Meta Modelling Framework," in *Proc. 1st International Fujaba Days*, H. Giese and A. Zündorf, Eds., vol. tr-ri-04-247, 10 2003, pp. 29–34.

[3] T. Y. Chen, P. L. Poon, and T. H. Tse, "A new restructuring algorithm for the classification-tree method," in *STEP '99*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 105–114.

[4] P. Clements and L. Northrop, *Software product lines: practices and patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[5] M. Conrad and A. Krupp, "An extension of the classification-tree method for embedded systems for the description of events." *Electr. Notes Theor. Comput. Sci.*, vol. 164, no. 4, pp. 3–11, 2006.

[6] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration through specialization and multilevel configuration of feature models," *Software Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005.

[7] ——, "Formalizing cardinality-based feature models and their specialization," in *Software Process: Improvement and Practice*, 2005, pp. 7–29.

[8] K. Czarnecki, S. She, and A. Wasowski, "Sample spaces and feature models: There and back again," in *SPLC '08*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 22–31.

[9] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: There and back again," in *SPLC '07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 23–34.

[10] G. E. F. (GEF). [Online]. Available: http://www.eclipse.org/gef/

[11] M. Grochtmann and K. Grimm, "Classification trees for partition testing," *Software Testing, Verification and Reliability*, vol. 1993, pp. 63–82, 1993.

[12] M. Janota and J. Kiniry, "Reasoning about feature models in higher-order logic," in *SPLC '07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–22.

[13] B. D. Jules White and D. C. Schmidt, "Selecting highly optimal architectural feature sets with filtered cartesian flattening," *Journal of Systems and Software to appear*.

[14] 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 University Software Engineering Institute, Tech. Rep., November 1990.

[15] K. R. P. H. Leung and W. Wong, "Towards a more efficient way of generating test cases: Class graphs," in *APAQS '00: Proceedings of the The First Asia-Pacific Conference on Quality Software (APAQS'00)*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 285–293.

[16] S. Lützkendorf and K. Bothe, "Attributierte Klassifikationsbäume zur Testdatenbestimmung," *Softwaretechnik-Trends*, vol. 23, no. 1, 2003.

[17] J. D. McGregor, "Testing a software product line," Tech. Rep. CMU/SEI-2001-TR-022, 2001.

[18] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval, "Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis," in *RE '07. 15th IEEE International*, 2007, pp. 243–253.

[19] E. M. Olimpiew and H. Gomaa, "Model-based testing for applications derived from software product lines," in *A-MOST '05*. New York, NY, USA: ACM, 2005, pp. 1–7.

[20] S. Oster and A. Schürr, "Architekturgetriebenes Pairwise-Testing für Software-Produktlinien," in *Workshop SE '09: Produkt-Variabilität im gesamten Lebenszyklus*, March 2009.

[21] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[22] K. Pohl and A. Metzger, "Software product line testing," *Commun. ACM*, vol. 49, no. 12, pp. 78–81, 2006.

[23] pure::systems. [Online]. Available: http://www.pure-systems.com

[24] S. Reis, A. Metzger, and K. Pohl, "Integration testing in software product line engineering: A model-based technique," in *FASE*, 2007, pp. 321–335.

[25] A. Reuys, E. Kamsties, K. Pohl, and S. Reis, "Model-based system testing of software product families," in *CAiSE*, 2005, pp. 519–534.

[26] K. Scheidemann, "Verifying families of system configurations," *Doctoral Thesis*, vol. TU Munich, 2007.

[27] I. Stürmer, H. Dörr, H. Giese, U. Kelter, A. Schürr, and A. Zündorf, "Das MATE Projekt visuelle Spezifikation von MATLAB Simulink/Stateflow Analysen und Transformationen," Dagstuhl Seminar Modellbasierte Entwicklung eingebetteter Systeme, Januar 2007.

[28] A. Tevanlinna, J. Taina, and R. Kauppinen, "Product family testing: a survey," *ACM SIGSOFT Software Engineering Notes.*, vol. 29, pp. 12–12, 2004.

Model-driven Support for Source Code Variability in Automotive Software Engineering

Cem Mengi*, Christian Fuß[†], Ruben Zimmermann[†], and Ismet Aktas[‡]

*Computer Science 3 (Software Engineering)
RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
Email: mengi@i3.informatik.rwth-aachen.de

[†]Carmeq GmbH
Carnotstr. 4, 10587 Berlin, Germany
Email: {christian.fuss | ruben.zimmermann}@carmeq.com

[‡]Computer Science 4 (Distributed Systems)
RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
Email: ismet.aktas@cs.rwth-aachen.de

***Abstract*—Variability on source code level in automotive software engineering is handled by C/C++ preprocessing directives. It provides fine-grained definition of variation points, but brings highly complex structures into the source code. The software gets more difficult to understand, to maintain and to integrate changes. Current approaches for modeling and managing variability on source code do not consider the specific requirements of the automotive domain. To close this gap, we propose a model-driven approach to support software engineers in handling source code variability and configuration of software variants. For this purpose, a variability model is developed that is linked with the source code. Using this approach, a software engineer can shift work steps to the variability model in order to model and manage variation points and implement their variants in the source code.**

***Index Terms*—automotive software engineering; programming; model-driven engineering; variability modeling;**

I. Introduction

Today the automotive industry provides customers a lot of possibilities to individualize their products. They can select from a huge set of optional fittings, e.g., parking assistant, rain sensor, intelligent light system, and/or comfort access system. The possibility to configure individual vehicles leads to the situation that both OEMs (Original Equipment Manufacturers) and suppliers have to capture explicitly potential *variation points* in their artifacts [1]–[3].

Thereby, the existence of variation points range over the whole electric/electronic (E/E) development process. They are available in the requirements, system specification, architecture design, source code, but also in the test and integration phase. Beyond that, variation points arise also during production, operation and maintenance phase. This means that in the whole product life cycle for a vehicle which hold up approx. 20-25 years, there evolve various types of variation points [4], [5]. Therefore, artifacts of different phases in the development process have to be investigated in order to explore their specifics [3], [6].

This paper deals with variability on source code level. Here, we focus on the programming languages C/C++, because they are the most widely used languages in automotive software engineering. With about 51% C has the most portion followed by C++ with about 30%. Assembler comes with about 8% and all other languages are applied less than 5% [7].

Variation points are implicitly modeled by implementing C/C++ preprocessing directives. In this way, variable (conditional) compilation results in specific software variants. This approach allows fine-grained definition of variation points, but brings highly complex structures into the source code. The software gets more difficult to understand, to maintain and to integrate changes. The main reason for this is that a software engineer has no support on source code level beside the programming language. Particularly, the user has to deal simultaneously with *problem space*, *configuration knowledge*, and *solution space* [8]. If a huge number of variation points exists, knowledge about a valid configuration gets difficult. Furthermore, a software developer has to find out the scattered code and the dependencies of one variant manually which is also very hard and time consuming.

There exists a wide range of techniques and mechanisms for modeling and managing variability [2], [9]–[15]. Most of them handle variability on a higher abstraction level. Elements of reusability are primarily software components, or constructs of object-oriented programming such as classes and methods which are replaced for specific variants of software. A support for fine-grained specifications of variation points on source code level are provided by a few number of concepts and tools [16]–[20], but they do not consider the specific requirements for the automotive domain. Particularly, safety critical applications come under regular code reviews and therefore have high demands on source code quality. Consequently, readability and understandability of source code are of high importance, but the above mentioned existing solutions do not consider this sufficiently.

To close this gap, we propose a model-driven approach to support software developers in handling versatile source code and configuration of software variants. For this purpose, we have developed a concept to separate problem space, configuration knowledge, and solution space. The problem space includes a common cardinality-based feature model to capture and manage variability [10], [11]. Furthermore, it supports the possibility to configure a software variant. The configuration knowledge can subsequently be transformed to the solution space. The solution space contains the source code. Here, we use a view-based approach in order to display the current configuration and hide everything that do not belong to the configuration.

The paper is structured as follows: In Section II, we analyze preprocessing directives that can express variation points. Particularly, a detailed consideration will show where problem space, configuration knowledge, and solution space is integrated. Furthermore, the problems that we will treat will be described in detail by using an example. In Section III, we will describe our approach to solve the problems. Here, we explain the separation of problem space, configuration knowledge, and solution space and go into detail of the three parts. Section IV, contains a short description of our implementation approach. In Section V, we will check if we have solved the mentioned problems. Finally, Section VI will summarize the paper.

II. Analyzing Source Code Variability

In this section, we will investigate how variability can be expressed using C/C++ preprocessing directives. We will introduce an example in order to explain arising problems of this approach in more detail.

A. Expressing Variability with C/C++ Preprocessing Directives

The current approach to express variation points and to configure specific software variants is to apply C/C++ preprocessing directives. For this purpose, statements for conditional inclusions are used, e.g., **`#ifdef, #ifndef, #if, #elif, #else`** (see Figure 1) [21]. In the following, we will use *preprocessing block* or *block* as a synonym for complete preprocessing directives.

The `identifier` for **`#ifdef`** and **`#ifndef`** directives in Figure 1a and 1b is a point of variation, because depending on its evaluation the contained source code is either included for compilation or not.

In the same way, the `constant-expression` in **`#if`** and **`#elif`** preprocessing directives shown in Figure 1c and 1d is also a point of variation. If it is evaluated to nonzero, the appropriate part of source code is included for compilation, otherwise not. Note, that a `constant-expression` allows more complex arithmetic and logical expressions. In the following, we will use *block rule* or simply *rule* as a synonym for a constant expression.

```
#ifdef identifier
   ...
#endif
```

(a) **#ifdef** preprocessing directive.

```
#ifndef identifier
   ...
#endif
```

(b) **#ifndef** preprocessing directive.

```
#if constant-expression
   ...
#endif
```

(c) **#if** preprocessing directive.

```
#if constant-expression1
   ...
#elif constant-expression2
   :
#elif constant-expressionN
   ...
#else
   ...
#endif
```

(d) **#if, #elif, #else** preprocessing directive.

Fig. 1. Preprocessing directives to handle variation points before compilation.

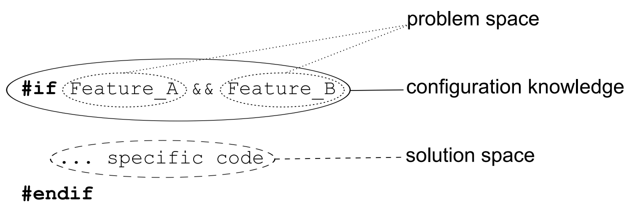


Fig. 2. Multilayer information in the solution space.

Analyzing preprocessing directives in detail, we have identified that different aspects of variability information is mixed into the code. We have decided to divide the information in analogy to Czarnecki's *generative domain model* which consists of a problem space, solution space and a configuration knowledge mapping between them [8]. Figure 2 illustrates this by an example.

The constant-expression `Feature_A && Feature_B` of the **`#if`** preprocessing directive is used to control the inclusion of the contained source code. Thereby, an identifier references a feature that is implemented in that code block, e.g., `Feature_A` and `Feature_B`. This kind of information is part of the problem space. The linking of an identifier with arithmetic and/or logical operations reflect configuration knowledge. Finally, the contained code reflect the implemen-

```
 1  #if PRIO_USE_SORTED_OBJECTS == 1
 2     #define PRIO_QUICKSORT 1
 3     #define PRIO_INSERTIONSORT 0
 4
 5     ...
 6
 7     #if PRIO_QUICKSORT
 8        ...
 9     #endif
10
11     #if PRIO_INSERTIONSORT
12        ...
13     #endif
14
15     static void sortTracks(...) {
16        #if PRIO_QUICKSORT
17           quicksortTrack(...);
18        #elif PRIO_INSERTIONSORT
19           insertionsortTracks(...);
20        #else
21           #error missing ...
22        #endif
23
24     }
25  #endif
```

Fig. 3. An example for variability handling with preprocessing directives.

tation which is part of the solution space.

B. Problem Description by Example

In this section, we will explain the problems that currently exists when dealing with C/C++ preprocessing directives to handle variability information. For this purpose, we will introduce an example.

Typically, sensors are adopted to collect data. In some situations it is necessary to prioritize the captured data. If so, different variants of sorting algorithms can be applied, e.g., quick-sort or insertion-sort.

The associated C source code is shown in Figure 3. The code between line 1 to 25 is only included if prioritization is selected. One of the sorting algorithms have to be configured (set to 1) in order to integrate the appropriate source code into the software variant. In our case, it is the quick-sort (see line 2). Particularly, the `sortTracks(...)` function (line 15) includes only the part of the source code which belongs to the quick-sort algorithm (line 17).

Although using preprocessing directives allows fine-grained and flexible specification of variation points, the source code gets more difficult to understand, to maintain, and to integrate changes. Analyzing the source code, we have identified four main problems, i.e.,

1) mixing problem space, configuration knowledge and solution space,
2) viewing all variation points without the knowledge of a valid configuration,

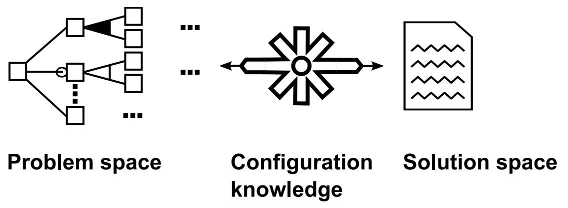


Fig. 4. Separation of problem space, configuration knowledge, and solution space.

3) code-variants of one variation point are scattered and have to be find manually, and
4) no explicit capturing of dependencies between variation points.

As described in Section II-A we have detected information in the source code that belongs to both problem space and solution space. For example, line 1, 7, 11, 16 etc. in Figure 3 are variability information that are part of the problem space and configuration knowledge. Even so, they are strongly integrated into the solution space, i.e., the source code.

Furthermore, considering the source code example, a software engineer always has to work with all variation points simultaneously, even most of them are not part of a specific variant. For example, the insertion-sort algorithm in Figure 3 does not belong to the variant if quick-sort is chosen (lines 3, 11-14, and 18-19). If more complex code sizes are regarded, solving a valid configuration gets more difficult.

Moreover, code-variants of one variation point are typically not implemented in a complete block but rather are scattered. For example, the quick-sort variant in Figure 3 appears in lines 2, 7-9, and 16-17. Particularly, this complicate including changes into code-variants or their appropriate preprocessing directives. If the code gets more complex, finding the code-variants manually gets very hard and time consuming. If changes into code or conditions have to be done, all relevant source code have to be find out manually to hold them consistent.

Finally, in many cases variation points are not isolated but depend on each other. In the source code, there is no explicit capturing of such information. For example, quick-sort and insertion-sort in Figure 3 are only included if a prioritization is necessary. If so, then they have an exclusive dependency on each other, i.e., only one can be chosen.

III. Model-driven Support for Source Code Variability

To deal with the identified problems mentioned in Section II-B, we propose a model-driven approach to treat source code variability and to support configuration of software variants. Therefore, we have developed a concept to separate problem space and configuration knowledge from solution space. The problem space is supported by a variability model that is based on Czarnecki's cardinality-based feature model [10], [11] (in the following we will use the term *variability model* as a synonym). Here, variation points are captured and

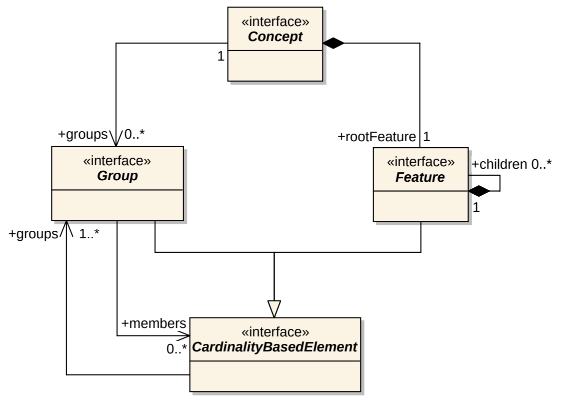


Fig. 5. Meta-model of the cardinality-based feature model.

managed. The configuration knowledge contains all informations to transform knowledge from problem space to solution space. The variability model supports the configuration. The solution space includes the source code. By integrating a view-based approach, only the configured part of the source code is displayed. This reflects the result from problem space transformed to solution space.

Figure 4 gives an overview of the separation of our approach. The general idea is, that a software developer not only work on the solution space, i.e., the source code, but also shift work steps into the variability model that is able to capture the problem space and configuration knowledge.

A. Source Code Variability Model

The focus on this paper does not lie on defining a new variability model, but rather using existing solutions to support variability on source code level. Analyzing existing approaches we have decided to adapt a cardinality-based feature model. Since it is a very common way to model variability, an integration of other tools and models get more simple. Particularly, this integration would allow using variability modeling techniques which are applied on a more abstract level, i.e., managing variability for classes, methods, objects etc. Our approach can then be used for fine-grained modeling of variability, i.e., on source code lines.

Figure 5 shows the meta-model for the cardinality-based feature model. It allows to define a tree-based structure. Thereby, a *Concept* node contains exactly one feature, i.e., the *rootFeature*. A Feature consists of an arbitrary number of *children* features. Moreover, a *Concept* node references an arbitrary number of *Groups* which define the number of elements in a group that can be specified for a configuration.

B. Transformation of Configuration Knowledge

To profit from the separation, it is an essential part to shift work steps to the central variability model. For this purpose, a connection between variability model and source code is necessary. To achieve this, we will use preprocessing directives which were, as described in Section II, the primary concept to express variation points. In this way, it will be possible to automatically add or delete preprocessing directives.

A user configures a specific variant whereas every modification of the source code, i.e., adding, deleting or modifying code lines, is linked with that configuration. Later on, it will be possible to display or hide code blocks depending on a specified configuration.

The basic principle for every transformation is the configuration knowledge from the variability model. If features *F_1, F_2, . . . , F_n* are selected, a transformation into a rule of the form `F_1 && F_2 && ...&& F_n` is executed. In the following examples, we always assume that this constant-expression is used.

Depending on the modification of the source code, the constant-expression is integrated into a preprocessing directive.

1) Modification of Source Code Outside Existing Preprocessing Blocks: The most simple case is when a software engineer modifies code outside existing preprocessing blocks.

a) Adding: If we have a rule of the form `F_1 && F_2 && ...&& F_n`, then it is embedded to an **`#if`** preprocessing block:

```
#if F_1 && F_2 && ...&& F_n
   ...
#endif
```

In this way, the code is only included for compilation, if the appropriate configuration is selected.

b) Deleting: Deleting code lines during a given configuration *F_1, F_2, . . . , F_n* do not delete them from the source file, but implies that the deleted lines should not appear in that configuration. For this purpose, we use the following **`#if`** preprocessing directive with the deleted code lines:

```
#if !(F_1 && F_2 && ...&& F_n)
   ... deleted code lines
#endif
```

2) Modification of Source Code Inside Existing Preprocessing Blocks: A slightly different case arises, if modifications inside existing preprocessing blocks are made.

a) Adding: If code is added inside a preprocessing block, then it is split in two blocks with the constant-expression as before and the added code lines are embraced with an **`#if`** preprocessing directive and a rule `F_1 && F_2 && ...&& F_n` that is transformed from the specified configuration.

```
#if constant-expression_1
   line_1
   line_2
   line_3
#endif
```

⇒

```
#if constant-expression_1
   line_1
#endif
#if constant-expression_2
   line_2
#endif
#if constant-expression_1
   line_3
#endif
```

In the example above, the blue marked code line (`line_2`) on the left side is added during a configuration *F_1, F_2, . . . , F_n*. In that case, `line_1` and `line_3` are embraced with the preprocessing directive as before and `line_2` is embraced with an **`#if`** preprocessing directive and a constant-

4 Implementation

5 Related Work

6 Conclusion

References

[1] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Trans. Softw. Eng.*, 34(2):162–180, 2008.

[2] T. Asikainen, T. Soininen, and T. Männistö. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. In *PFE 2003: Software Product-Family Engineering, 5th International Workshop*, volume 3014 of *Lecture Notes in Computer Science*, pages 225–249

[12] B. W. Kernighan and D. M. Ritchie. *C P guage 2nd Ed.* Prentice Hall, January 198

[13] F. J. v. d. Linden, K. Schmid, and E. Ro *Product Lines in Action: The Best Indu Product Line Engineering*. Springer-Verla Secaucus, NJ, USA, 2007.

[14] C. Lopes and G. Kiczales. Aspect-orien *Technology of Object-Oriented Langua Conference on*, 0:468, 2000.

[15] C. Mengi and I. Armaç. Functional Var Adaptable Functional Networks. In *VaMo ternational Workshop on Variability Mode Intensive Systems*, volume 29 of *ICB Rese* 83–92. Universität Duisburg-Essen, 2009.

[16] S. J. Paul, P. Bassett, H. Zhang, and W. Z based variant configuration language. I

expression F_1 && F_2 && ...&& F_n (in the figure above, denoted as constant-expression_2).

This adaptation differs from the transformation for modification of source code outside existing preprocessing directives. If we would transform the added code lines in the same way as in Section III-B1a then we would get a nested structure. But this would bring an implication into the code that possibly is not planned by a software developer. For example, if line_2 would be nested into the superior preprocessing block, then the code lines would only exist in a variant that includes a configuration of the superior block. By dividing them in multiple blocks of preprocessing directives this side effect is avoided and the described implication is still possible if the software engineer uses the configuration of the variability model.

b) Deleting: When deleting code lines, the mentioned problems for adding code do not appear, because the reference to the superior preprocessing block is mandatory and must be kept, so that the constant-expression of the superior preprocessing block and the constant-expression that is transformed from the current configuration must be included.

```
#if constant-expression_1
  line_1
  ~~line_2~~
  line_3
#endif
```

⇒

```
#if constant-expression_1
  line_1
#endif
#if constant-expression_1
    && !constant-expression_2
  ~~line_2~~
#endif
#if constant-expression_1
  line_3
#endif
```

In the example above, the red marked and struck out code line (line_2) on the left side is deleted during a configuration $F\_1, F\_2, \ldots, F\_n$. In that case, line_1 and line_3 are embraced with the preprocessing block as before and line_2 is included into an **#if** preprocessing block with a constant-expression constant-expression_1 && !constant-expression_2, where constant-expression_2 is the result of the transformation of the current configuration, i.e., F_1 && ... && F_2. We have decided to split the preprocessing block but nesting them would in this case also be possible.

If only **#if** constant-expression_1 would be included, then the deleted code line would be appear in each variant that do not contain the configuration $F\_1, F\_2, \ldots, F\_n$. Particularly, it would be independent from constant-expression_1.

3) Modification of Source Code for Complete Preprocessing Blocks: Beside of adding or deleting code lines, in some situations it is also reasonable to add or delete complete preprocessing blocks in a given configuration.

a) Adding: If it is necessary to add a preprocessing block of one variant (or configuration) into another one, this could be done by configuring the variant where the code block appears, copying it, configuring the variant where it should appear, and then pasting it. This method is a little bit uncomfortable. Therefore, we support adding complete code blocks into a configuration automatically without copy/paste actions. For this purpose, we only have to extend the constant-expression of the preprocessing block which include the code lines that should appear in the current configuration.

```
#if constant-expression_1
  line_1
  ...
  line_n
#endif
```

⇒

```
#if constant-expression_1
    || constant-expression_2
  line_1
  ...
  line_n
#endif
```

In the example above, the blue marked code lines on the left side should included into a configuration $F\_1, F\_2, \ldots, F\_n$. The appropriate preprocessing block after transformation is shown on the right side. The code line now would appear in configuration that is transformed to constant-expression_1 or constant-expression_2, where constant-expression_2 is the current configuration $F\_1, \ldots, F\_n$.

b) Deleting: If a complete preprocessing block is deleted, an adaptation of the constant-expression should be made. Thereby, the transformation of a configuration is motivated by the same principle as for deleting code lines inside existing preprocessing blocks.

```
~~#if constant-expression_1~~
  ~~line_1~~
  ...
  ~~line_n~~
~~#endif~~
```

⇒

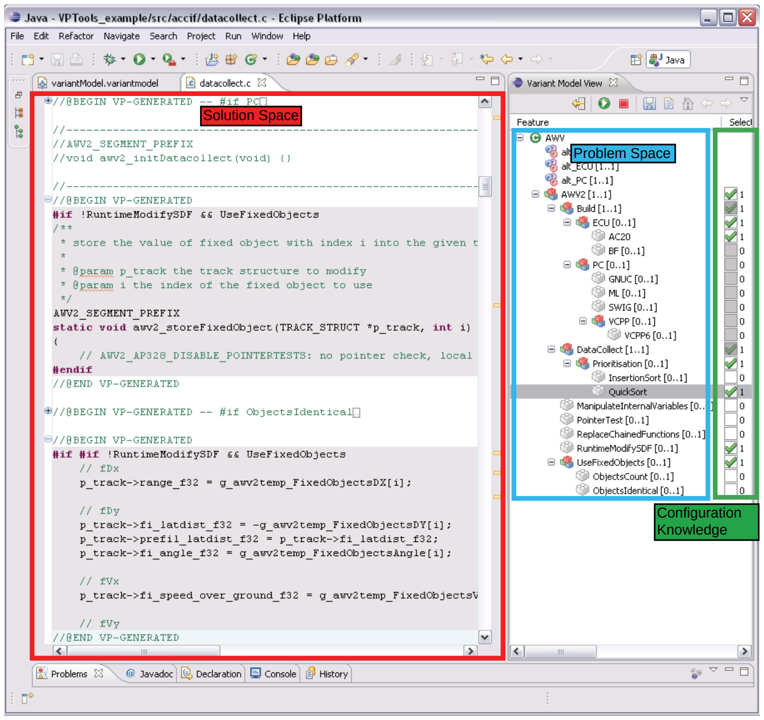
```
#if constant-expression_1
    && !constant-expression_2
  line_1
  ...
  line_n
#endif
```

In the example above, the red marked and struck out code lines on the left side are deleted during a configuration $F\_1, \ldots, F\_n$. The result of the transformation is shown on the right side. Considering the constant-expression on the right side, the code lines only appear, if constant-expression_1 holds and constant-expression_2 is equal to F_1 && ... && F_n.

4) Variant Source Code: The solution space of our approach is the source code. In order to take the advantages of the division, the transformation of configuration knowledge must be reflect in the source code. For this purpose, we have adopt a view-based approach where all source code is hidden that is not part of the configuration. The configuration is made on the cardinality-based feature model, which was explained in Section III-A. Depending on the configuration, all constant-expressions of preprocessing directives are evaluated to decide whether the block should be displayed or hidden. In principle, this emulates the preprocessor with the advantage that targeted configurations can be viewed.

IV. Implementation

The described concepts are implemented in a way that they can be integrated into existing development processes and projects as seamless as possible. Therefore, we had to follow general requirements:


Java - VPTools_example/src/accif/datacollect.c - Eclipse Platform
File Edit Refactor Navigate Search Project Run Window Help
Java
variantModel.variantmodel
datacollect.c
Solution Space
Variant Model View
Feature
Select
Problem Space
Configuration Knowledge
Problems
Javadoc
Declaration
Console
History

```
//@BEGIN VP-GENERATED -- #if PC

//----------------------------------------------------------------------
//AWV2_SEGMENT_PREFIX
//void awv2_initDatacollect(void) {}

//----------------------------------------------------------------------
//@BEGIN VP-GENERATED
#if !RuntimeModifySDF && UseFixedObjects
/**
 * store the value of fixed object with index i into the given t
 *
 * @param p_track the track structure to modify
 * @param i the index of the fixed object to use
 */
AWV2_SEGMENT_PREFIX
static void awv2_storeFixedObject(TRACK_STRUCT *p_track, int i)
{
    // AWV2_AP328_DISABLE_POINTERTESTS: no pointer check, local
#endif
//@END VP-GENERATED

//@BEGIN VP-GENERATED -- #if ObjectsIdentical

//@BEGIN VP-GENERATED
#if #if !RuntimeModifySDF && UseFixedObjects
    // fDx
    p_track->range_f32 = g_awv2temp_FixedObjectsDX[i];

    // fDy
    p_track->fi_latdist_f32 = -g_awv2temp_FixedObjectsDY[i];
    p_track->prefil_latdist_f32 = p_track->fi_latdist_f32;
    p_track->fi_angle_f32 = g_awv2temp_FixedObjectsAngle[i];

    // fVx
    p_track->fi_speed_over_ground_f32 = g_awv2temp_FixedObjectsV

    // fVy
//@END VP-GENERATED
```

```
AWV
  alt
  alt_ECU [1..1]
  alt_PC [1..1]
  AWV2 [1..1]                              ✓ 1
    Build [1..1]                           ✓ 1
      ECU [0..1]                           ✓ 1
        AC20                               ✓ 1
        BF [0..1]                            0
      PC [0..1]                              0
        GNUC [0..1]                          0
        ML [0..1]                            0
        SWIG [0..1]                          0
        VCPP [0..1]                          0
          VCPP6 [0..1]                       0
    DataCollect [1..1]                     ✓ 1
      Prioritisation [0..1]                ✓ 1
        InsertionSort [0..1]                 0
        QuickSort                          ✓ 1
    ManipulateInternalVariables [0..1]       0
    PointerTest [0..1]                       0
    ReplaceChainedFunctions [0..1]           0
    RuntimeModifySDF [0..1]                ✓ 1
    UseFixedObjects [0..1]                 ✓ 1
      ObjectsCount [0..1]                    0
      ObjectsIdentical [0..1]                0
```

Fig. 6. A screenshot of the developed Eclipse plugin.

1) At all time, valid C/C++ code must be available.
2) Editing and maintenance of source code must be possible without the need for specific tooling.
3) Additional work load for a software developer must be as low as possible.
4) Dynamic changes must be feasible.

The implementation is fulfilled by developing a plugin for the *Eclipse Framework* [22]. The cardinality-based feature model is implemented with support of the *Eclipse Modeling Framework (EMF)* [23]. An editor for the feature model was also generated by using EMF which can be used in parallel to the Eclipse C/C++ Development Tooling (CDT) [24]. A screenshot is shown in Figure 6.

The left part contains the editor where C/C++ source code can be written. The right part contains a view on a configurable feature model. The software developer can use both parts in parallel in order to configure a specific variant of interest so that all other code lines that are not included into the variant are hidden. In some situations, not all modifications on model configuration should influence the view on the source code. In the same way, not all modifications on source code should influence the selected configuration. For this purpose, the user gets the possibility to explicitly select a control element that triggers the linking between code and model. If the linking is stopped the transformation is subsequently executed. This means, that all preprocessing directives are added into the source code.

The editor to configure a variant has the ability to select or deselect features and to solve implications. Furthermore, the configuration of invalid variants are avoided. At the same time, it supports a software developer to detect modeling errors.

V. Problems Revisted

If we consider again the listed problems in Section II-B, we observe that they are solved by the described concepts.

The core problem was that problem space, configuration knowledge, and solution space were mixed. By dividing them we have formed a basis to solve the other problems. Knowledge about a valid configuration is given through support of the configurable feature model. Code-variants must not find out manually, but are solved by the configuration which then is transformed to the source code. By adopting a view-based approach only the relevant code lines are displayed. Dependencies between variation points are also stored in the feature model by expressing cardinalities.

Overall, complex work steps are now shifted to a model where they can be handled more easier. The software engineer can now concentrate on the main work, i.e., developing software.

VI. Conclusion

In this paper we have described a model-driven approach to handle source code variability. We have outlined existing problems, analyzed them in detail in order to propose a solution. The main problem is that problem space, configuration knowledge, and solution space is mixed, i.e., a software engineer works only on the source code without any support to treat variability. This leads to the situation that source code is overcrowded with variation points without knowing how they depend on each other. In our approach we have suggest a division of problem space, configuration knowledge, and solution space. A cardinality-based feature model is adopted and linked with the source code in order to shift work steps into the model. By a configuration support modifications on the source code are linked with the model. Furthermore, transformation of configuration is supported by adopting a view-based approach.

In future work, we want to integrate this approach with earlier phases of an E/E development process. Software architectures are one essential artifact that need support for variability handling. If variability support is provided, an integration with the source code level would be an essential benefit.

References

[1] P. Clements and L. Northrop, *Software product lines: practices and patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[2] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, September 2005.

[3] F. J. v. d. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[4] M. Broy, "Challenges in automotive software engineering," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 33–42.

[5] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, "Software engineering for automotive systems: A roadmap," in *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 55–71.

[6] C. Mengi and I. Armaç, "Functional Variant Modeling for Adaptable Functional Networks," in *VaMoS 2009: Third International Workshop on Variability Modelling of Software-Intensive Systems*, ser. ICB Research Report, vol. 29. Universität Duisburg-Essen, 2009, pp. 83–92.

[7] Embedded Systems Design website, http://www.embedded.com/.

[8] K. Czarnecki and U. Eisenecker, *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

[9] 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 University Software Engineering Institute, Tech. Rep., November 1990.

[10] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[11] K. Czarnecki and C. H. Kim, "Cardinality-Based Feature Modeling and Constraints: A Progress Report," in *OOPSLA'05 International Workshop on Software Factories*, October 2005.

[12] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, "COVAMOF: A Framework for Modeling Variability in Software Product Families," in *SPLC 2004: Software Product Lines, Third International Conference*, ser. Lecture Notes in Computer Science, vol. 3154. Springer, 2004, pp. 197–213.

[13] D. Beuche, H. Papajewski, and W. Schröder-Preikschat, "Variability management with feature models," *Sci. Comput. Program.*, vol. 53, no. 3, pp. 333–352, 2004.

[14] Pure Systems website, http://www.pure-systems.com/.

[15] T. Asikainen, T. Soininen, and T. Männistö, "A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families," in *PFE 2003: Software Product-Family Engineering, 5th International Workshop*, ser. Lecture Notes in Computer Science, vol. 3014. Springer, 2003, pp. 225–249.

[16] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 311–320.

[17] S. Apel, T. Leich, and G. Saake, "Aspectual feature modules," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 162–180, 2008.

[18] C. Lopes and G. Kiczales, "Aspect-oriented programming," *Technology of Object-Oriented Languages, International Conference on*, vol. 0, p. 468, 2000.

[19] A. Bryant, A. Catton, K. De Volder, and G. C. Murphy, "Explicit programming," in *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2002, pp. 10–18.

[20] S. J. Paul, P. Bassett, H. Zhang, and W. Zhang, "Xvcl: Xml-based variant configuration language," in *ICSE '03: Proceedings of the international conference on Software engineering*, 2003, pp. 810–811.

[21] B. W. Kernighan and D. M. Ritchie, *C Programming Language, 2nd Ed*. Prentice Hall, January 1988.

[22] The Eclipse Foundation website, http://www.eclipse.org/.

[23] F. Budinsky, S. A. Brodsky, and E. Merks, *Eclipse Modeling Framework*. Pearson Education, 2003.

[24] Eclipse C/C++ Development Tooling Project, http://www.eclipse.org/cdt/.

Aspect-Oriented Patterns for the Realization of Flexible Feature Binding

Kwanwoo Lee
Department of Information Systems Engineering
Hansung University
Seoul, Korea
kwlee@hansung.ac.kr

***Abstract*—Feature selection is the process of determining features that should be included in a product to satisfy the requirements for the various stakeholders. Feature binding time refers to the time at which variable features are selected for a product and their implementations are bound into the product. A feature may have different binding times for different products. In this paper, we present an aspect-oriented approach to supporting flexible feature binding time.**

***Index Terms*—component; formatting; style; styling;**

I. INTRODUCTION

In software product line engineering, a product is derived by selecting some of product line features that satisfies the product requirements. Feature selection is the process of determining features that should be included in a product to satisfy the requirements for the various stakeholders.

The time at which features are selected for a product may vary depending on marketing strategies. For instance, one marketing strategy may be to deliver products to customers by prepackaging product specific features, as customer needs in this market rarely change. In this case, product specific features may be selected for a product during product build time. On the other hand, another marketing strategy may be to allow customers to start with a product with core features and then grow to a bigger one by adding new features at load time or run time. In this case, product specific features may be selected for products at product load time or run time.

Feature binding time refers to the time at which variable features are selected for a product and their implementations are bound into the product. Feature binding time has significant influences on the way a feature is implemented. There are many variability mechanisms for realizing feature binding times. Some of those include conditional compilation, macro processing, virtual dispatch tables, reflection, dynamic class loading, etc.

These mechanisms, however, are strongly tied to a particular choice of binding time [1]. The problem may occur when a feature may have different binding times for different products. That is, the variability of feature binding time affects the way a feature is implemented. To support multiple feature binding times effectively, code for feature binding times needs to be separated from feature implementations.

Aspect-oriented programming (AOP) provides effective mechanisms for separating crosscutting concerns from modular components. Since the code for multiple feature binding times may affect multiple feature implementation components, this paper uses AOP mechanisms to achieve flexible feature binding time. That is. The approach makes it possible to choose among compile-time, load-time and run-time binding for selected features.

For better understanding of this paper, the next section presents the concept of feature binding. Based on this understanding, aspect-oriented patterns for supporting flexible feature binding time are presented in section 3. Section 4 discusses areas requiring further research and concludes this paper.

II. FEATURE BINDING

A feature has to be bound into a product to provide its capability to users. As shown in Fig. 1, an unbound feature must be first included into a product to provide its capability. However, it is considered that a feature is not bound into a product, if it is not available to users although included in the product. Therefore, feature binding means that a feature is included into a product and become available to users. It is important to note that available features that are bound into a product can provide their capabilities to users only when they are active.

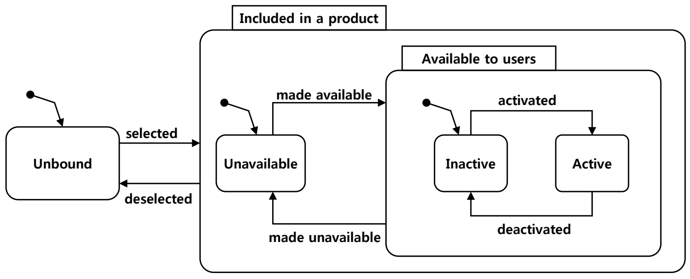


Fig. 1. Feature Binding Context.

For feature binding to occur, the inclusion and availability of a feature may be decided simultaneously at compile time, load time, or run time. Or, the inclusion of a feature is decided at compile time, but its availability may be postponed until load time or run time.

Feature binding at compile time: In this case, the code related to the selected features is included into a product and becomes available to customers at compile time, while the code for deselected features is excluded from the product. This results in different packaging of the product.

Feature binding at load time: There are two cases for load time feature binding to occur. The first case is that features are included into a product at compile time and become available at load time. The second case is that the inclusion and availability of features are decided at load time. The main difference between the compile feature binding and the load time feature binding is that the software derived from the compile time feature binding must be compiled, whereas that from the load time feature binding needs not.

Feature binding at run time: Feature binding in this class is similar to the load time feature binding. That is, features are included into a product at compile time and become available at run time or included at product load time and available at run time. Alternatively, both the inclusion and availability of features are decided at run time.

Depending on the inclusion and availability decisions of a feature, it may have to be implemented in different ways. In the next section, we present an aspect-oriented approach to supporting flexible feature binding times.

III. Aspect-Oriented Design Patterns

The current AspectJ weaver provides explicit support for compile-time and load time weaving. This implies that if we implement variable features with aspects, we do not need to change the aspects to support for either compile-time or load-time feature binding. However, when we want to implement a load-time binding feature that has to be included at compile-time and becomes available at load-time, we have to change the feature implementation to support the required feature binding decisions. Moreover, if we consider run-time feature binding, we may have to change the feature implementation as well. This implies a feature with multiple binding times may have to be implemented differently depending on which binding time decisions are decided for a product.

In this section, we present aspect-oriented design patterns for supporting flexible feature binding times.

A. Variable Inclusion Decisions

For feature binding to occur, the code implementing a feature must be included in a product and integrated with the other code for the product. Since the time at which a feature is included in a product may vary for different products in a product line, the variable inclusion times may require variable feature implementations.

To support flexible binding times effectively, the code implementing the decision on feature inclusion needs to be separated from the code implementing the core functionality of a feature. As shown in Fig. 2, `CompileTimeBinding` is an aspectual implementation for integrating the module (`VFModule`) implementing a variable feature with the module (`CFModule`) in the scope of a product at compile time. The pointcut `variationPoint` defines the join points in `CFModule` at which `VFModule` is bound into a product. The advice body in `CompileTimeBinding` defines actual binding between `CFModule` and `VFModule`. If `VFModule` and `CompileTimeBinding` are given to a AspectJ compiler at compile time, the compiler produce a weaved product.

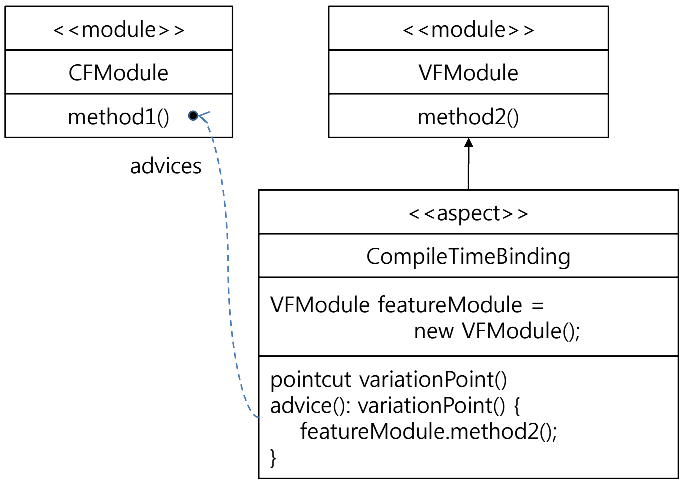


Fig. 2. Compile-Time Inclusion Pattern.

Note that AspectJ provides explicit support for load time weaving. Therefore, we can integrate `VFModule` and `CompileTimeBinding` with an existing product at load time, simply by including them before program execution.

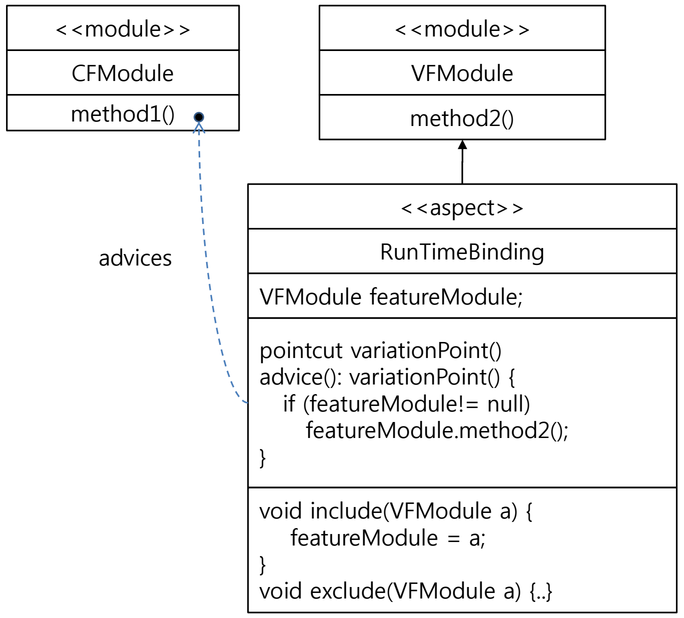


Fig. 3. Run-Time Inclusion Pattern.

Although AspectJ does not provide support for run time weaving, we can implement the run time feature binding which includes the implementation of a variable feature in a product at run-time. As shown in Fig. 3, `RunTimeBinding` allow `VFModule` implementing a variable feature to be included in a product scope and integrated with `CFModule`, which is an implementation module in the product scope.

`RunTimeBinding` is similar to `CompileTimeBinding` in that it specifies the join points used to integrate `CFModule` and `VFModule` using pointcut definitions. However, their integration is deferred until the actual instance of `VFModule` is included in the product at run-time. A external configurator has the responsibility for including the instance using the `include` method based on the user's decision at run-time.

B. Deferring Availability Decisions

Although AspectJ weaver does not provide explicit support for run-time weaving, we can support runtime feature binding using AOP, in case inclusion is determined at compile-time but availability is decided at load-time or run-time. That is, availability of included aspects is decided by enabling or disabling advices in the aspects.

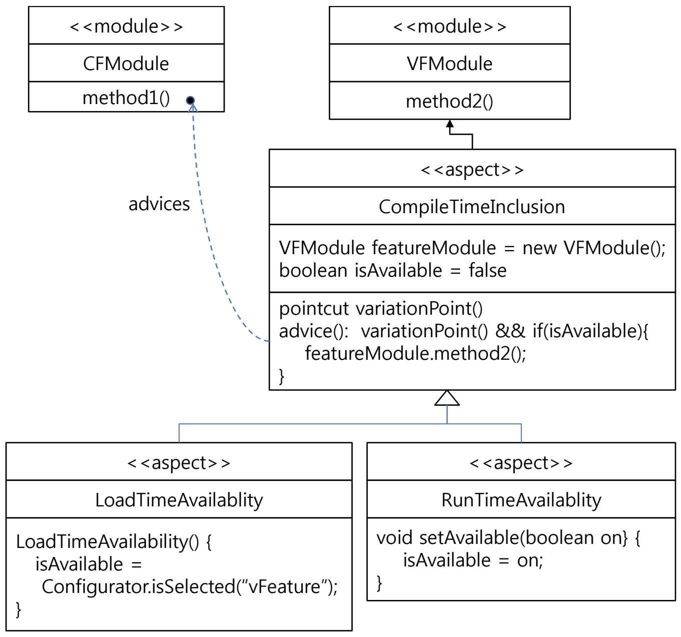


Fig. 4. Variable Availability Pattern.

Fig. 4 shows how to support the load-time or run-time binding of a variable feature, which is included at compile time but becomes available during load-time or run-time. `CompileTimeInclusionNotAvailable` is similar to `CompileTimeBinding` shown in Fig. 2 except that the boolean type variable `isAvailable` is used to allow the availability decision to be decided in its child aspects (`LoadTimeAvailability` and `RunTimeAvailability`).

`LoadTimeAvailability` determines the value of `isAvailable` when the aspect is created by consulting with an external configuration, which has responsibility for configuring a product after compilation. On the other hand, `RunTimeAvailability` sets the variable `isAvailable` to true using the method `setAvailable`. But the decision is made by an external configurator at run-time.

With these patterns, we can clearly separate binding time decisions from the code implementing the core functionality of a feature. This enables us to select different choices among multiple feature binding implementations when a feature has multiple feature binding times.

IV. Related Work

The concept of feature binding time was first introduced by Kang et al. [3]. Gurp et al. [5] elaborated it more precisely and provided a classification of many variability realization techniques. A broad range of mechanisms exist to implement different binding times, including the use of compiler directives, dynamic linking and loading, load tables, reflection, plug-ins, configuration files, etc. However these solutions are limited in that each supports only a specific binding time. They cannot be used effectiely in a situation where the binding time of a feature may vary depending on different product requirements.

Dolstra et al. [2] introduced the notion on the variability of feature binding time as *timeline variability*. However, they do not provide concrete mechanisms for realizing flexible feature binding time. They only suggested some future directions for the solutions.

There have been several attempts to realize flexible feature binding time. Hoek [6] proposed architecture-based approach to support any-time variability. The Koala component model [7] allows connection between components to be established either at compile time or at run time through *a switch*. Depending on the setting of a switch, the Koala compiler generates C code for connecting components either at compile time or at run time. Both of these approaches specify product line variabilities at design time, but resove them at any time thereafter. On the other hand, the approach presented in this paper uses aspect-oriented design patterns using AOP.

Similar to our approach, Edicts [1] uses AOP for flexible feature binding. However, it only supports run time feature biding which includes a feature at compile time and makes it available at run time. But our approach can add more flexibility of feature binding time from different choices of inclusion and avaiability decisions.

V. Research Issues and Conclusions

In this paper, aspect-oriented patterns are introduced to support flexible feature binding times. There are many issues that have to be addressed before this approach becomes useful. Some of these issues are summarized below:

- *Method*: We need methods for analyzing feature binding time, developing a software product line applying the patterns presented above, deriving a product based on a feature configuration, which can be determined at either compile-time, load-time, or run-time.

- *Feature model extension*: Although there have been many attempts to extend the original feature model [3], there has been no attempt to model the variability of feature binding time. Since the inclusion and availability decisions for feature binding are affected by resources available during run time or development environments such as programming languages or operating environments, when we model variabilities of feature binding time, we may take into account constraints or dependencies from various sources (e.g., available resources). Also we may have to consider finer classification of feature binding times.
- *Implementation mechanism*: In this paper, we illustrated the patterns using AspectJ. However, more advanced mechanisms such as Prose [4], which supports run-time weaving, can be used to support flexible feature binding time. Which one among current available technologies or mechanisms can be best utilized for achieving this goal? We need to analyze pros and cons for each technology or mechanism.

In this section, we have examined some of research issues that have to be addressed. Although we are at an early stage of research, most of research topics discussed above are being addressed.

Acknowledgment

This work was supported by the Korea SW Industry Promotion Agency (KIPA)

References

[1] V. Chakravarthy, J. Regehr, E. Eide, *Edits: Implementing Features with Flexible Binding Times*, AOSD'08, 2008.

[2] E. Dolstra, G. Florijin, E. Visser, *Timeline Variability: The Variability of Binding Time of Variation Points*, Proceedings of the Workshop on Software Variability Management, 2003, pp. 119-122.

[3] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, *Feature-Oriented Domain Analysis (FODA) feasibility study*, Software Engineering Institute Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[4] A. Popovici, T. Gross, G Alonso, *Dynamic Weaving for Aspect-Oriented Programming*, Proceedings of the 1st international conference on Aspect-oriented software development, Enschede, The Netherlands, 2002, pp. 141-147

[5] M. Svahnberg, J. van Gurp, and J. Bosch. *A taxonomy of variability realization techniques*, Software-Practice & Experience, 35(8), pp. 705-754, 2005.

[6] A. van der Hoek, *Design-Time Product Line Architecture for Any-Time Variability*, Science of Computer Programming, Vol. 53, Issue 3, December 2004, pp. 285-304

[7] R. van Ommering, *Building Product Populations with Software Components*, ICSE'02, May 19-25, 2002, pp.255-264