Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line

Stefan Ferber[1], Jürgen Haag[2], and Juha Savolainen[3]

[1] Robert Bosch GmbH, Corporate Research and Development FV/SLD
Eschborner Landstraße 130–132, D-60489 Frankfurt, Germany
Stefan.Ferber@de.bosch.com
[2] Robert Bosch GmbH, Gasoline Systems GS/ESK
P.O. Box 30 02 40, D-70442 Stuttgart, Germany
Juergen.Haag2@de.bosch.com
[3] Nokia Research Center
P.O. Box 407, FIN-00045 NOKIA GROUP, Finland
Juha.Savolainen@nokia.com

Abstract. Reengineering a legacy product line has been addressed very little by current product line research activities. This paper introduces a method to investigate feature dependencies and interactions, which restricts the variants that can be derived from the legacy product line assets. Reorganizing the product line assets with respect to new requirements requires more knowledge than what is easily provided by the classical feature-modeling approaches. Hence, adding all the feature dependencies and interactions into the feature tree results in unreadable and unmanageable feature models that fail to achieve their original goals.
We therefore propose two complementary views to represent the feature model. One view shows the hierarchical refinement of features similar to common feature-modeling approaches in a feature tree. The second view describes what kind of dependencies and interactions there are between various features.
We show two examples of feature dependencies and interactions in the context of an engine-control software product line, and we demonstrate how our approach helps to define correct product configurations from product line variants.

1 Introduction

Automotive products are rich with variants to meet the special needs of different customers. Applying the Product Line Approach (PLA) for the embedded systems in automotive products is therefore quite natural. A product line is an important system development paradigm in the automotive industry to amortize costs beyond a single product. The paradigm is well established in the mechanical and electrical engineering practice in automotive companies like Bosch. To make the car more secure, economical, clean, and comfortable, more functionality is moving from mechanical to electrical, and from electrical to software solutions. Therefore, today's automotive products are software-intensive systems that are developed with the PLA paradigm [6,9,17,29,31].

Feature modeling provides an industrial strength method for managing the inherent variability of product lines. The variability modeling is often done using two techniques:

G. Chastek (Ed.): SPLC2 2002, LNCS 2379, pp. 235–256, 2002.

tables and graphical models. In this paper, we concentrate on graphical modeling using feature trees and interaction diagrams. Based on our experience, graphical representation allows the easy understanding of complex structures and their dependencies, which is especially important in the reengineering context. On the other hand, using tables does not require any special tools, and it provides easy processing of information by automatic tools. In fact, these methods are complementary. We often model the first version of variability by listing all entries into a table. Later, we collect the variability information into a feature tree and use the graphical representation as a basis for creating the dependency and interaction models. Structuring features in trees allows to model requirements in different granularities. In practice, it is difficult to model features with all their implications and dependencies with current feature-oriented methods. In the real-life context, relations between features often become very complex without a clear way to model features with different dimensions or aspects, leading to a very complex graph of features. These graphs have nodes that represent the features and different types of edges between the nodes where the edges connect nodes in other parts of the feature hierarchy. The feature-oriented domain analysis (FODA) method [19] and successors of FODA [7] require a tree-like structure of the feature model which may be very difficult to generate, as there are no clear criteria to align the structure to a tree-like form. Moreover, you cannot capture all of the information contained in a (general) graph in a tree, leaving you with the possibility of omitting dependencies or losing control of the model[17].

Feature dependencies and interactions play an important part in complex software-intensive systems, since features often need other features to fulfill their tasks. There are feature dependencies, which are imposed by the domain, and other dependencies that exist due only to a certain design decision made during development or to the way that features are realized in the solution. Categorizing types of interactions by distinguishing all possible feature dependencies in a domain greatly helps to reengineer product line assets. This is because some domain-enforced dependencies among features are difficult if not impossible to change while dependencies that are based on design decisions can often be removed during the reengineering process. The feature-dependency analysis is a crucial part of the product line development, where the features are used to distinguish between product line variants. In this context, feature interaction can have very strong implications on the possible configurations of product line members. Knowing and managing the dependencies becomes very important, especially in highly configurable product lines.

We suggest different views on the feature model like architectural views on the architecture to represent the desired information. In this paper, we show two examples of feature views. One view captures the FODA-like concept of hierarchical refinement of features and their variability assumptions. In the second view, we present (static and dynamic) feature interactions and dependencies. This dependency and interaction view was used to reengineer features from legacy specifications and implementations of a product line.

The next section of this paper reviews current feature-modeling methods and motivates our extension, which is described in detail in Section 3. In Section 4, we apply our method to two examples. Our findings are related to other research in this area as

described in Section 5. Finally, in Section 6, we conclude our experience and give an outlook on the future.

2 Feature Modeling

Features bridge the marketing view to the development view. On the marketing, side features define properties that the customer wants from the system. On the development side features are used as increments of implementation. Typically, features are defined in terms that the customer can understand. The customer focus also supports the use of features for configuration purposes, where the customer can choose which features are included in a product variant. The configuration aspect of the feature-based development is even more important when the product line techniques are used.

In the product line context, features are used to differentiate product line members. The high-end products tend to have more features than inexpensive product variants. Additionally, some features for more complex products may have more complex behavior. To handle this variation in features, a number of different approaches have been proposed. Currently, one of the most successful approaches in variability modeling is to use feature trees. The feature tree is a feature-refinement hierarchy that explicitly shows the variability among the product line members by attaching different symbols that communicate the related variability assumptions.

However, modeling variations among features is not enough to make the feature model an efficient tool for product derivation. The features are not independent from each other. All the features work together to achieve a common goal: the purpose of the system. In the gasoline engine-control domain, this is realizing a pleasant, economical driving experience that also addresses environmental values. The features cannot achieve this without interchanging information interacting among each other. The traditional justification of modeling software features is to see them as independent entities that can be specified and hopefully developed in isolation. But often, the crucial part of specifying a complex software system is actually based on the interactions of the features. This revelation justifies the clear need to model dependencies and interactions among software features. The interaction-modeling community has produced a lot of high-quality research on how the combined behavior of features can be specified and validated [33]. Specifying behavioral aspects of the software is beyond the scope of this paper, since we only address feature relationships as they affect the configuration and derivation of product variants. In this context, a key aspect of understanding the various relationships between two features is to realize how they are dependent on each other. To support this, some feature-modeling methods suggest a way to model dependencies among features. Most notably, in the FODA method, various rules are formulated to represent the dependencies. These rules, for example, set and require conditions between two features. Unfortunately, these rules make it very difficult to get the big picture of the overall dependencies. The other option is to add dependency links directly into the feature trees making them into more complex graphs. Modeling dependencies in this way as a part of the feature tree greatly reduces the understandability of the feature model, and degrades their usefulness. Even though some methods explicitly address the feature dependency problem, a surprisingly large number of feature-modeling methods

do not provide any way to model dependencies among features. In the real-life context, feature trees may become very large. Any attempt to model dependencies among features as a part of the feature tree is doomed to failure in industrial use because of the lack of mechanisms to handle scalability. We solve these problems by providing another view on features: the feature-dependency view. For more discussion on approaches for modeling dependencies see Section 5.

The feature-dependency model describes all possible dependencies among features within the current scope. It shows how the functionality of features is dependent on the implementation of pieces of other features' functionality. In this paper, we use feature dependency for two main purposes. First, we record the feature dependencies to better understand the legacy product line by identifying how the various features are interrelated. This insight assists us in our reengineering efforts to create a new product line architecture. The feature dependencies are key information that can be used when a new reference architecture is created. In many systems, well-known functionality-related criteria, such as cohesion and coupling, are used to determine the correct system decomposition. Identifying the types of relationships seems to be a crucially important aspect, while identifying possible transformations of the existing architecture.

Second, dependency modeling is used to allow the easy derivation of product instances from the reference architecture. In the real-life product lines that we have experience with, the architecture is not driven only by the desired properties of the products; an equally important aspect is to identify what is possible based on the solution domain. Often, certain features are cheaper to implement using services from other features. Sharing common components may be appropriate even when some of the components' services are not needed for the low-end product. This kind of sharing of common assets may create dependencies between features that, from the customer's viewpoint, are not needed at all. Yet, the product derivation clearly requires this information while creating the low-end product. We solve this problem by expressing these kinds of relationships between features in the feature-dependency view.

In this paper, we show these techniques in the context of reengineering. But the models are also useful when creating new systems. However, then the engineers should guarantee that they do not prematurely commit on the specific designing choices while analyzing dependencies among features. These considerations are out of the scope of this paper, and all the recommendations of this paper are targeted on using the approach for reengineering existing product lines.

3 Two Views of the Feature Model

This paper assumes that a feature model describes complex interactions and dependencies between features in a product line context, which involves a lot of variability issues. Moreover, features have additional attributes like a detailed description, current state of development (e.g., specified, coded, tested, released), delivery information, needed memory and processor resources, traces to software components, and many more. Therefore, the complete feature model can hardly be described in one diagram or in one table. There are three possible ways to deal with the complexity in such models:

1. Split the model into multiple diagrams with different features in each diagram.

2. Use a hierarchy in which features can be described in more detail with subfeatures.
3. Employ different views with each view dealing with a certain aspect of the model.

We are using all three approaches to analyze the features in the software of a control system for gasoline engines, as we have to cover several hundred features with complex relationships (see Section 4). This paper covers only two views: one showing variability, the other expressing dependencies. But, we can think of additional views in this application: feature road maps and feature resource consumption. Probably, other domains require their own views.

3.1 Used Terminology

A feature is "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems" [19]. A domain model is "a definition of the functions, objects, data, and relationships in a domain" [19]. And a feature model is a specific domain model covering the features and their relationships within a domain. The domain in this case is the product line scope of the engine-control software for gasoline engines called motronic.

A hierarchy of features with semantic edges as relationships is called a feature tree. The edges can represent alternative, multiple, optional, and mandatory features. Two features, B and C, interact if the responsibility of feature B is to change the behavior of feature C. Usually, these features have a runtime relationship to fulfill their responsibility. Feature B depends on feature C, if feature B can fulfill its responsibility only if feature C is present (i.e., B requires C), or if feature C is not present (i.e., B excludes C).

The feature model consists of all features and relationships among them. The feature-tree view shows the variability assumption. The feature interaction and dependency view shows all feature interaction and dependency relationship. Note that some information is present in both views. For example, if feature B excludes feature C, one can represent this in the tree with an alternative relationship and in the dependency and interaction view with a relationship called "excludes".

The feature-tree view is the appropriate tool to structure features of a product line. Moreover, it can be used to select features for a product that is derived out of the product line even by non-expert users of a software system (e.g., salespeople, customer). The feature interaction and dependency view helps software integrators and developers to identify valid feature configurations in the product line. We used both views to mine and document the assets of the motronic product line.

3.2 Syntax and Semantics in the Feature Interaction and Dependency View

In this section, we describe how feature interactions and dependencies can be modeled and recorded for documentation and derivation use. In our model, we connect the features that participate to the interaction by using arrows that represent the types of the interactions. For the motronic domain, we have chosen five main types of feature interactions: usage, intentional, resource usage, environment induced, and excluded relation.

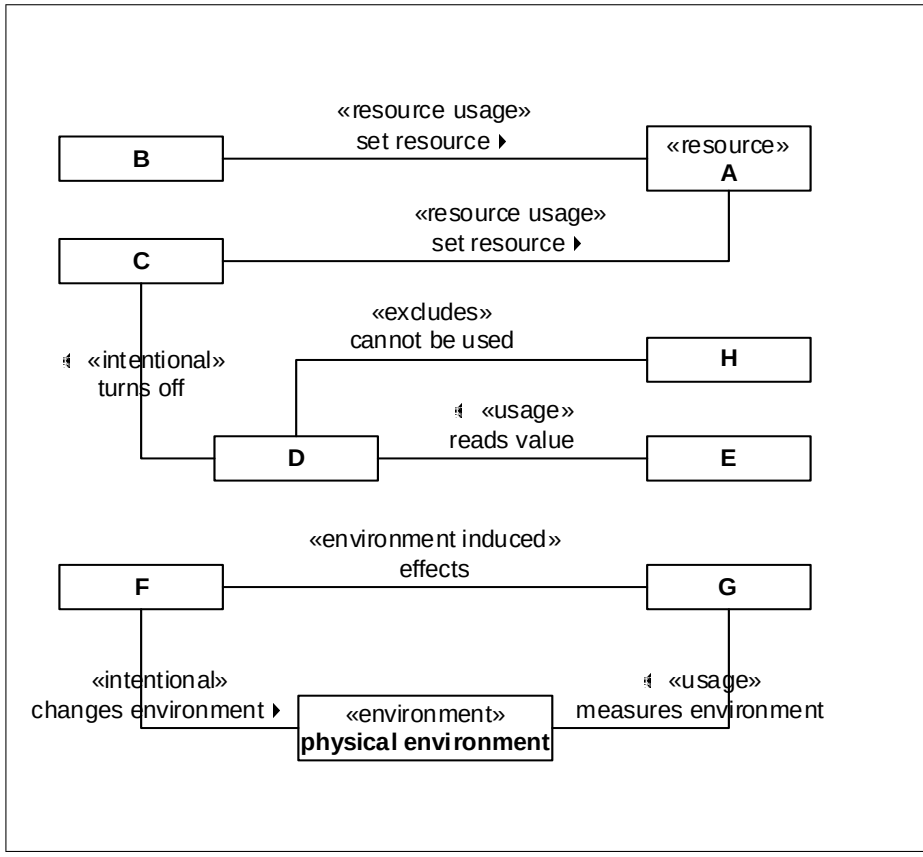

Fig. 1. Feature Interaction and Dependency Representational Model

Intentional Interaction. For some features, a large part of their functionality is based on the nature of the interaction, that is, they encapsulate a piece of interaction-oriented functionality. These features typically change the behavior of other features. For example, in the engine control software, the "limp-home" features influence the synchronization of the crankshaft and camshaft angles in certain degraded mode conditions (see the example in Figure 5).

Resource-Usage Interaction. Some resources are limited in the sense that only a subset of all features can simultaneously use them. Typically, this kind of relationship also presents itself in the timing-dependency-relationship dimension. Often this interaction is taken care of by another feature or service that manages the interaction between a potentially large number of features. For example, the driver requests high torque to accelerate the car, but the automatic transmission requests a low torque to shift down. A feature called Torque Coordinator schedules the torque demands from different sources.

Environment Induced Interaction. There are also other kinds of indirect, often very implicit, connections between features. These occur when a set of features monitor some properties of the physical environment on which some other features have some influence. In the concrete instances of this kind of relationship, it is important to specify how the features affect the environment. This means that features that are not directly connected together are affected by the changes in the environment. We record it as an implicit, environment-induced interaction between the corresponding features. These kinds of dependencies are difficult to notice, since most of the time there is no direct connection between the features. For example, the fuel-purge control adds gasoline into the intake manifold, which is observed much later by the exhaust lambda sensors.

Usage Dependency. The most common form of feature dependency is the usage relationship. This can be a simple "requires" relationship. Data-exchange relationships are normally based on the usage dependency. If a feature uses data from another feature, it creates a one-directional usage dependency from the data consumer to the data producer. Quite naturally, if we have multiple features, we can easily end up with cyclic relationships between the corresponding features. For example, sensor diagnostics for the camshaft sensor reads values provided by the Sense Camshaft Angle and Sense Crankshaft Angle features to detect inconsistent sensor values (see the example in Figure 5).

Excluded Dependency. Some features cannot be in the same family variant after the binding time. The excluded relation is heavily dependent on the binding time of the features: for example, if the binding time and the run time can both be present in the actual implementation of the family variant, they cannot operate at the same time. E.g. At one time there can be only either camshaft limp-home mode or the crankshaft limp-home mode active, since either one sensor is needed for the engine speed calculation. This definition means that the limp-home feature is excluded at runtime. Thus both features can exist in one family variant, but only one of them can be active at the same time. There is a natural reason for this kind of behavior, since either the crankshaft or camshaft sensor must be operational to guarantee speed calculations. If neither of these sensors functions correctly, no limp-home features can be activated (see the example in Figure 5).

Out-of-Scope Stereotype. Feature interaction and dependency, which crosscut views, are modeled in detail in only one view. Other views refer to these features with the "out-of-scope" stereotype.

3.3 Syntax and Semantics of the Feature-Tree View

Domain assumptions reveal the knowledge that a developer has on the operating domain. Domain assumptions constrain how product family members may vary within the current scope. The possible domain assumptions are mandatory, optional, alternative, and multiple.

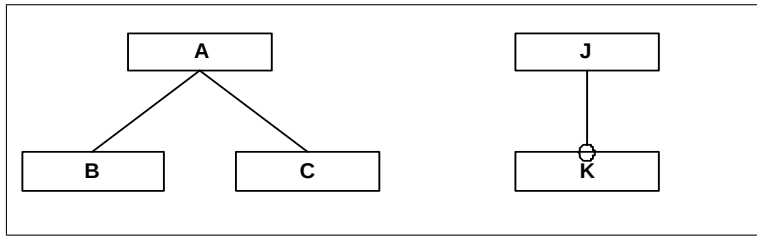

Fig. 2. Feature Variability Representational Model in the Feature Tree: The subfeature relationship is shown on the left and the optional feature is shown on the right.

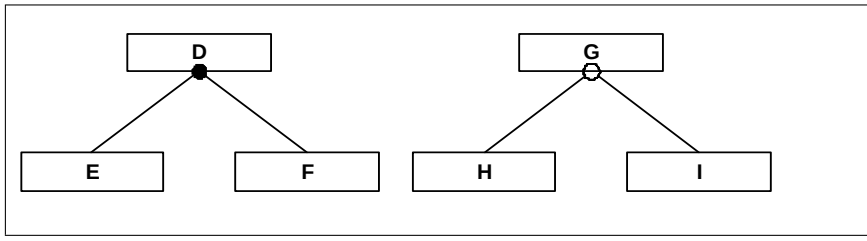

Fig. 3. Feature Variability Representational Model in the Feature Tree: The multiple relationship is shown on the left and the alternative relationship is shown on the right.

In Figures 2 and 3 we show all possible variability types and their corresponding representations. A solid line between features defines a subfeature relationship where child features are mandatory, if their parent is chosen. This is the case between features A, B, and C. Features E and F are part of a multiple relationship where one or both of the features can be chosen. On the other hand, H and I are alternative features, one of which must be chosen. Finally, K is an optional feature.

Mandatory: If, for a set of features, we have a mandatory domain assumption, then these features have to be present in all the products that belong to the current scope. For example, every family variant has an ignition system.

Optional: Optional domain assumption describes a feature that may or may not be included into the family variant. An optional domain assumption does not constrain the available options in any way it merely specifies that it is likely that, within the current scope, there will be systems that have this particular feature and those that do not. For example, an engine may have a turbo-charger.

Alternative: An alternative domain assumption represents a choice between a set of features. One and only one feature has to be chosen from the alternative set of features. For example, throttle-valve control is either by air-bypass actuator, throttle-valve actuator, or electronic throttle control.

Multiple: Multiple domain assumption captures a possibility to choose many features from a set of features, but the user has to choose at least one. For example, an engine can have an intake manifold pressure sensor, an air mass flow meter, or both.

3.4 Consistency of Views

Showing one model in two views raises a question of their consistency. We use very simple rules to achieve practical consistency in our models. Rules for the feature-tree view are

1. All features of the feature model are presented in the tree view (completeness).
2. Variability assumptions in the tree view are a subset of all dependencies (incomplete but free of contradiction).

Rules for the feature interaction and dependency view are

1. Only a subset of features are presented in the interaction and dependency view (incomplete).
2. All dependencies and interactions are shown in the interaction and dependency view (completeness).

These rules satisfy the practical needs for consistency, since in our process, we first create a feature tree, which has all the features that reside in the current domain. After that, we analyze dependencies. In the dependency view, we have a subset of all features, but we cannot have any features that are not part of a feature tree. If the feature is not in the feature tree of the related domain, then it is tagged with the stereotype “out of scope”. That is, there cannot be any feature that is not defined in a feature tree. The dependency view adds information on how those features interact, but it cannot add new features without also adding them into the feature trees.

4 Feature Modeling for Engine Control Software

The feature-modeling method introduced earlier in this paper was used to analyze the functionality of the software of the Motronic control system for gasoline engines [13,14].

The Motronic system contains all of the actuators (servo units, final-control elements) required for intervening in spark-ignition engine management, while monitoring devices (sensors) register current operating data for engines and vehicles. Some of these sensor signals are

– Accelerator-pedal travel,
– Engine position,
– Cylinder charge factor (air mass),
– Engine and intake-air temperatures, and
– Mixture composition (air-fuel ratio, called “Lambda”).

The system, consisting of a microprocessor and software, employs these data as the basis for quantifying driver demand, and responds by calculating the engine torque required for compliance with the driver’s wishes. Out of the required engine torque, the actuator signals for

- electronic throttle device (and therefore for the air mass),
- injection (fuel mass),
- ignition,

and other devices are calculated. By ensuring the provision of the required cylinder charge together with the corresponding injected fuel quantity, and the correct ignition timing, the system furnishes the optimal mixture formation and combustion to fulfill the driver's wishes as well as emission laws.

Subsequently, two examples of the feature analysis are shown. The first example is the engine position management (EPM) of the motronic. In this example, the feature tree was created first to identify the current features and its variations. The feature interaction diagram was developed afterwards to identify the dependencies between the features.

The second example shows the feature analysis of the control system for the air-fuel ratio of the gasoline engine's exhaust gas (Lambda feedback control). In this example, the interaction diagram was used to rearrange the tree view of the feature model.

4.1 Example 1: Engine Position Management

The Engine Position Management (EPM) of the control system for gasoline engines has different responsibilities. A feature tree of the control system gives an overview of the functionality in the EPM domain.

Figure 4 shows the main features of the EPM. First of all, the engine speed and the crankshaft angle have to be sensed. But also the camshaft angle sensing and the synchronization are major parts of the EPM. A reliable synchronization in the whole speed range of the engine is necessary for the right ignition and injection timing and therefore to fulfill the strict emission laws. The optional limp-home modes (in case of sensor failure) and the start-up functionality are subfeatures of the synchronization. The start-up of the engine can be realized in different ways: fast, fast with history, or reliable. These subfeatures have a "multiple" relationship because a combination of them is required by some customers.

The "back-rotation detection" feature is optional because only some engine types could be destroyed if the engine runs in the wrong direction. This feature has two "alternative" subfeatures: it can either use the camshaft or the crankshaft sensor for detection.

There are more dependencies between the features of the EPM than shown in the feature tree. Of course these dependencies have to be known to reengineer features when implementing the EPM for a new product line. To describe all the dependencies, the interaction diagram (introduced in Section 3) was developed for the EPM (see Figure 5).

Though there are many dependencies and interactions between features in this case, they do not restrict the selection in the tree view to derive a valid product out of the product line features. You can see in Figure 6 that, for example, minimal feature selection does not contradict the feature interaction and dependency view.

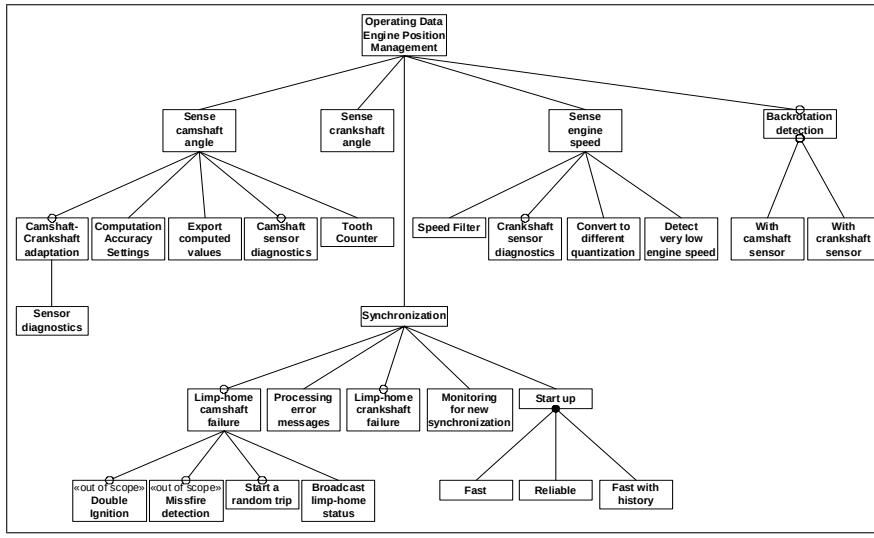

Fig. 4. Feature Tree View of the Engine Position Management (EPM).

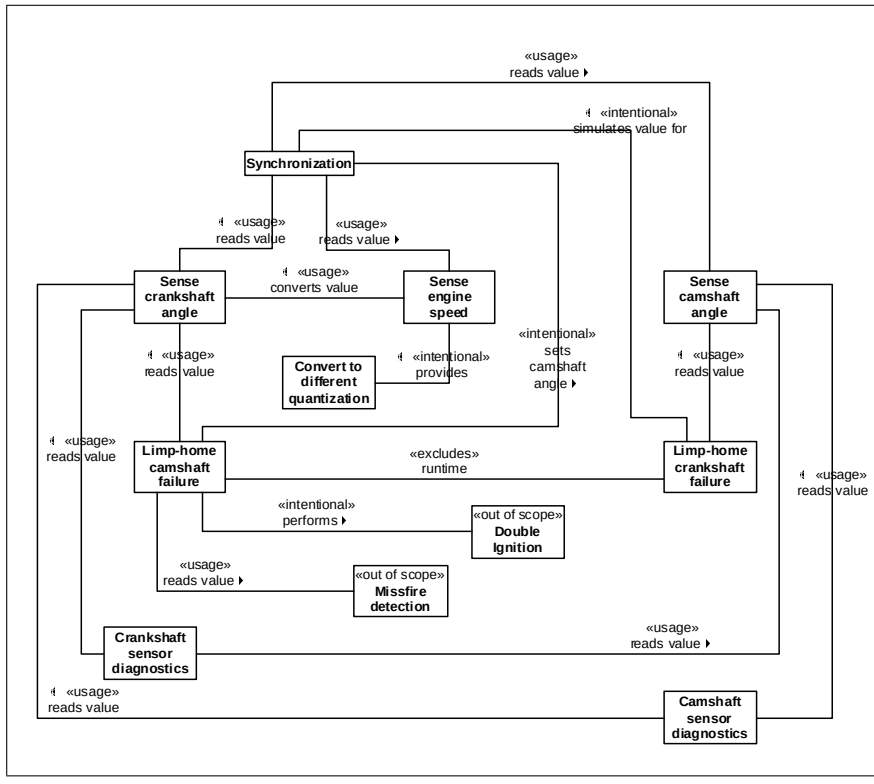

Fig. 5. Feature Interaction and Dependency View of the Engine Position Management (EPM)

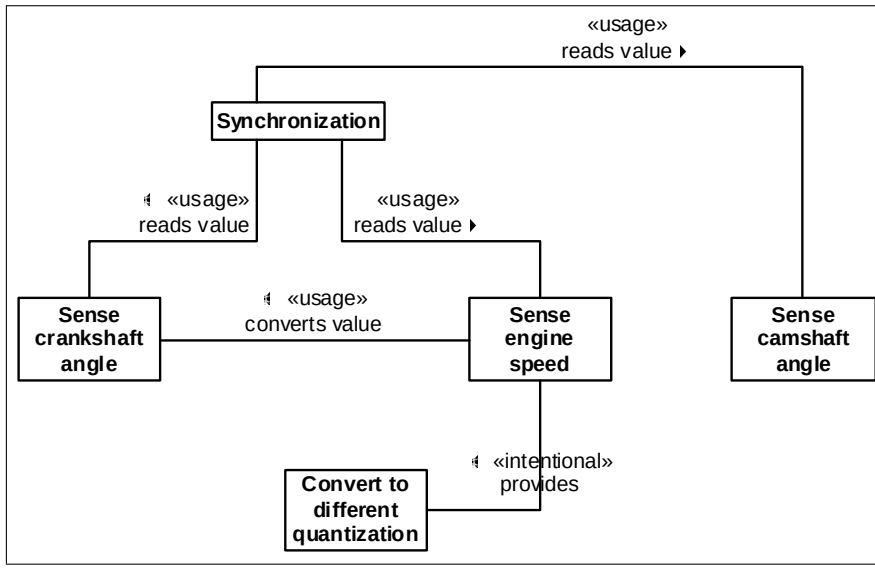

Fig. 6. Minimal Configuration for the Engine Position Management in the Feature-Tree View. It is also valid in the interaction and dependency view.

4.2 Example 2: Lambda Feedback Control

In this example, the control of the air-fuel ratio (Lambda feedback control) of the exhaust gas was investigated. Due to the strict emission laws, the complexity of the Lambda control system has increased dramatically during the last few years. Therefore, different control strategies and several sensor configurations are used to fulfill the requirements.

Figure 7 shows the possible configurations of the lambda sensors and the catalysts in the exhaust system. The first sensor can be a two-state lambda sensor or a wide-band lambda sensor. All other sensors are two-state lambda sensors. The precatalyst and the middle sensor are optional, depending on the emission requirements. There are several control strategies for the lambda control:

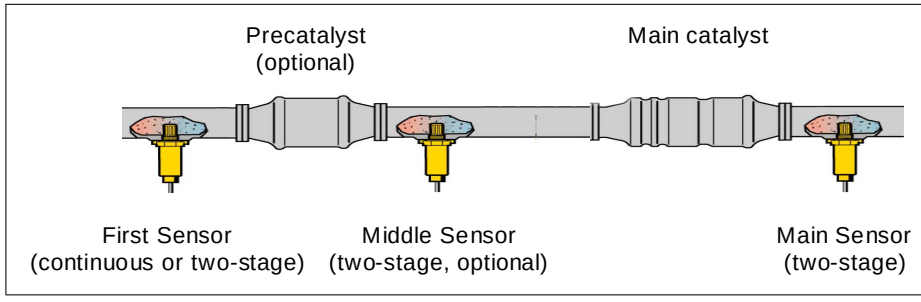

Fig. 7. Naming of Sensors and Catalysts in the Exhaust System

– continuous or two-state lambda feedback control first sensor
– continuous lambda feedback control middle sensor
– continuous lambda feedback control main sensor
– natural frequency feedback control
– catalyst outcome feedback control

Even for the domain expert, it was difficult to get an overview of the possible sensor configurations in combination with these different control strategies and their dependencies. Mixing features with the current implementation of these features made things even more difficult. This was probably due to the history of the functionality. As you can see, this feature tree (Figure 8) has two main disadvantages. First, the structure of the

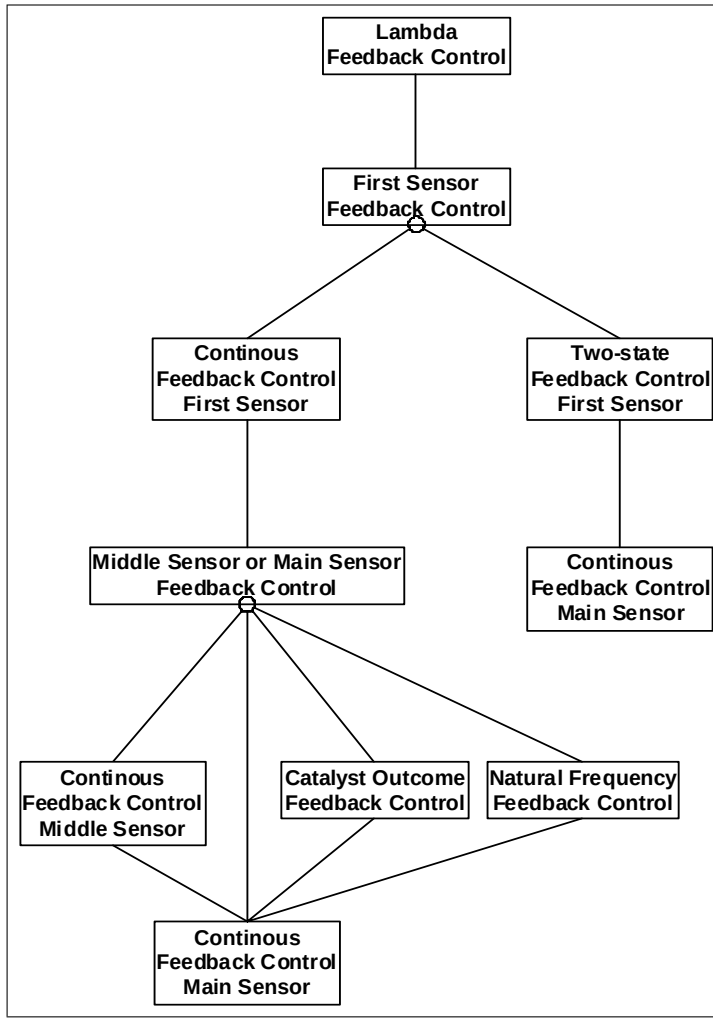

Fig. 8. Feature "Tree" of Exhaust System Feedback Control Before Analyzing Dependencies

tree isn't a proper tree structure, and second, it is necessary to represent the "continuous feedback control main sensor" feature twice.

To restructure the feature tree, the dependencies and interactions between the features were modeled. To express indirect dependencies, the related sensors were added in the same diagram (Figure 9). Due to the modeled dependencies (especially "exclude" and "requires") between features and sensors, it was possible to derive a very simple new feature tree containing most (except one) of the dependencies in Figure 10.

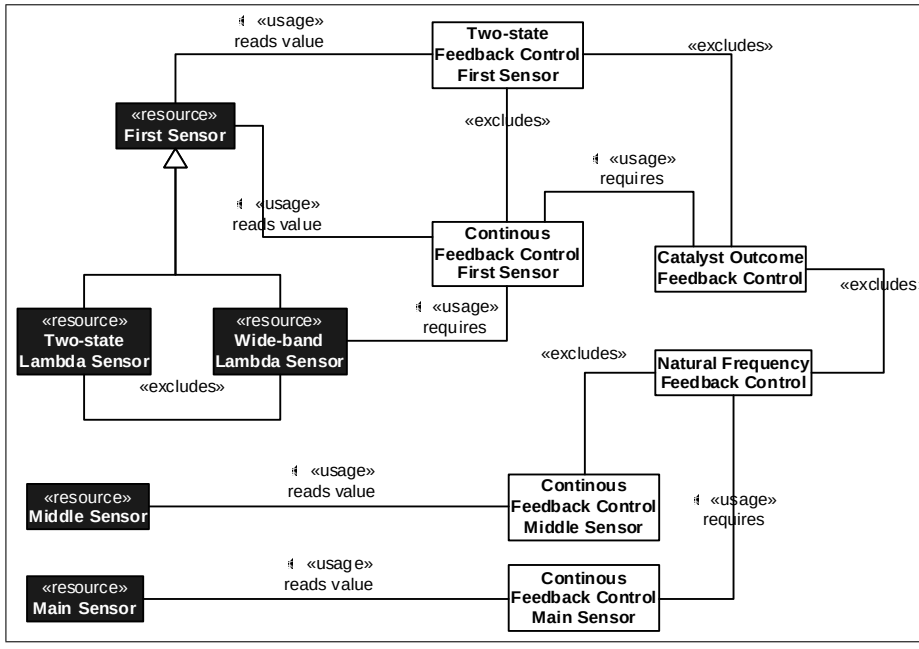

Fig. 9. Feature Dependency and Interaction View of the Lambda Feedback Control

Still, one needs the information in the interaction and dependency view for deriving products in the product line. Figure 11 shows a "valid" configuration of the feature-tree view in Figure 10. But, the exclude relationship of two selected features is only covered in the interaction and dependency view. In order to have a valid configuration, all the information in the feature model is needed.

4.3 Tracing the Realization of Features

In this feature analysis case, the software is already there. Capturing the traces for each feature is therefore quite easy. We collected additional information for each feature:

1. The variability mechanism for alternative, multiple, or optional features
2. The name of the variability point in the software
3. The software function that provides the feature

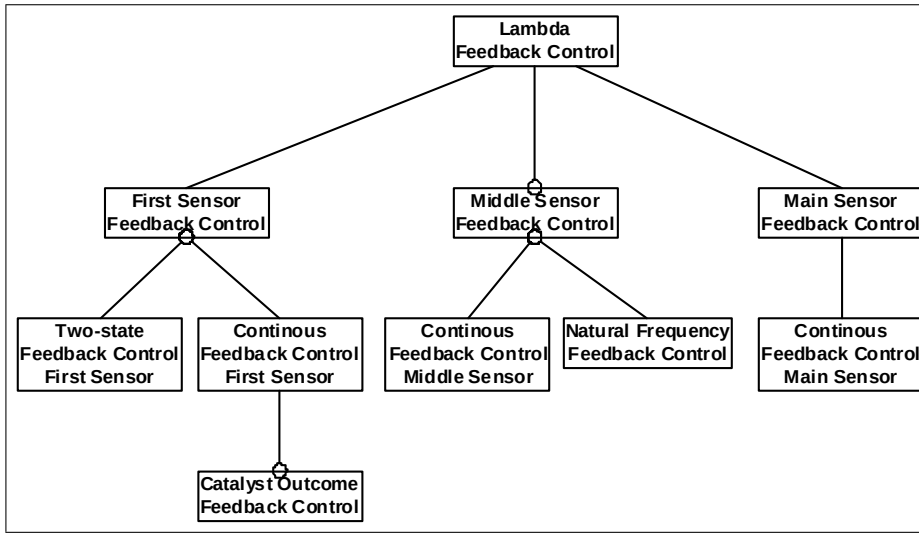

Fig. 10. Rearranged Feature-Tree View After Analyzing Dependencies of the Lambda Feedback Control

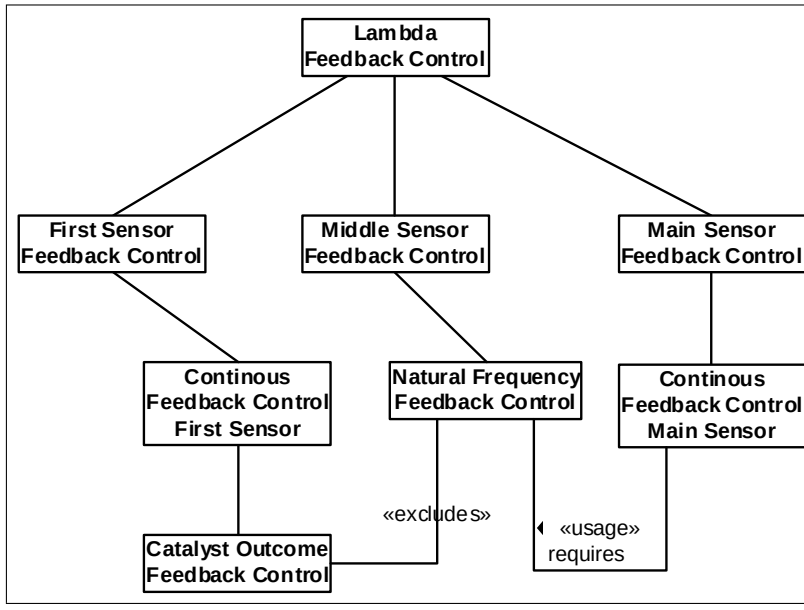

Fig. 11. An invalid configuration of lambda feedback control for one product in the product line.

This was captured in UML-like style as shown in Figure 12.

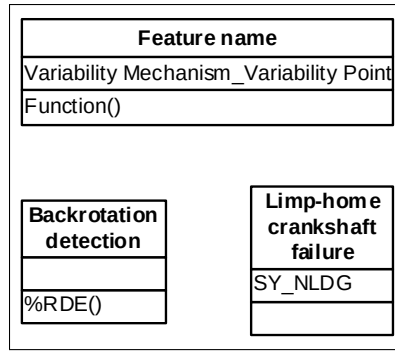

Fig. 12. Tracing Features to Variability Points in the Software and to the Software Function: The upper box describes the generic feature trace template, and the two lower boxes are examples of its usage for the EPM.

4.4 Modeling the Complete System

We used this approach to model the complete engine-control software. The feature model is shown in 40 different feature-tree views. We have shown only two simplified versions in this paper. Each feature-tree view has between 50 and 200 features and describes the functionality of a conceptual object in the domain. The depth of the hierarchy varies from three to nine whereas a depth of five seems to be a good tradeoff between overview and detail.

From this feature model, we were able to extract high-level and crosscutting features that describe the basic nature of the system. We call these features *system features*. There are nearly 300 system features which cover the most important feature dependencies.

Our experience indicates that this approach scales well. Maintaining the trees was the most difficult and time-consuming problem, as we had to rearrange the features by hand. Currently, an XML-based feature model with a set of supporting tools is under development and will (hopefully) resolve this problem.

5 Related Work

Feature-modeling research has been an integral part of product line research for many years. This has resulted in a natural tendency to explicitly target variability issues during the method development. Each of the methods has some way of representing the commonality and variability in the product family. However, surprisingly large number of feature-modeling methods do not provide equally powerful techniques to model dependencies between features. Especially, they do not provide any tools to manage the inherent complexity of the dependency models in any way that can support the scalability

needed in the current industrial projects. In this section, we will focus on how existing feature-modeling methods model dependencies among features.

In the widely used feature-oriented domain analysis method, rules are created to represent the dependencies [7,19]. Composition rules define the semantics existing between features that are not expressed in the feature diagram. All optional and alternative features that cannot be selected when the named feature is selected must be stated using the mutually exclusive WITH statement. All optional and alternative features that must be selected when the named feature is selected must be defined using the REQUIRES statement. Unfortunately, these rules make it very difficult to get the big picture of the overall dependencies. The only way that approaches the solution of the scalability issues is the FODA approach, with its feature rules. However, we believe that there are major drawbacks to this approach. Separating the rules from the actual features that are affected by the rule makes it necessary in larger projects to search the whole set of rules to find those that have some effect on the current feature in question. This separation reduces understandability, hinders evolution, and complicates derivation.

On the variability management view, the FODA approach is quite similar to ours with its three kinds of structuring constructs. Mandatory features are those requirements of the system that have to be included in the product family. Alternative features are specializations of more general features that have multiple specialization features, one of which has to be chosen. Optional constructs model features that can be included into one product, but may be left out of another product. Our model shares these definitions on the three types of variability.

The Feature-Oriented Reuse Method (FORM) is an extension to the FODA approach [23]. FORM and FODA use similar notations for feature modeling. In the FORM method, feature diagrams support three kinds of relationships: a composed-of relation that represents a whole-part relationship between a feature and its children; a generalization/specialization relationship represented by the feature refinement structure; and an implemented-by relation where one feature is used to implement another feature. In addition to the feature relationships that conform to the feature-tree structure, the FORM method also supports composition rules that constrain the possible feature selection across feature-tree branches. The composition rules define mutual exclusion and mutual dependency among optional and alternative features in manner very similar to that used in FODA method.

Griss and colleagues have proposed integrating feature modeling as used in the FODA method into the Reuse-Driven Software Engineering Business (RSEB) method [15]. In the RSEB method, the domain is specified in the RSEB use-case model, which uses the standard methods for modeling relationships applying uses and extends relationships together with variation points among use cases[1]. The feature model is used to configure the use-case model, by expressing which use cases can be combined, selected, and customized for this system. The method also uses semantic constraints like REQUIRES and MUTUAL EXCLUSION. These rules constrain the selection of the optional and variant features much as the FODA method does. They suggest adding the constraints to

[1] Note that the current UML notation actually uses extend, include, and generalization relationships.

the UML diagrams as attributes or as UML constrains added to the classes or relationships (e.g., in the object constraint language (OCL)).

The generative programming framework by Czarnecki and Eisenecker also applies feature modeling for organizing knowledge of the product family structure [8]. They have extended the FODA feature models with OR features. They also provide methods and guidelines for normalizing feature diagrams, which basically means ways of transforming diagrams to another consistent model with the same set of features. Additionally, they indicate the need to identify constraints and dependency rules among features. The focus of their work concentrates on defining mutual-exclusion and REQUIRES constraints, as defined in the FODA method. In addition to previously published methods they also distinguish between horizontal and vertical constraints, which mainly define whether the features sharing a constraint belong to the same or different abstraction level. This work does not provide any explicit tools to define or mange constraints, but it implies that the OCL can be used to specify different constraints.

Feature modeling and analysis have been proved to be highly important aspects of modern industrial product line development. Thus, company-specific, feature-classification schemes have also emerged. The large-scale use of feature-modeling techniques lead to problems with feature dependencies, which do not fit into the chosen domain decomposition. The modeling of features' coupling crosses feature-tree branches and transforms feature trees into graphs that may become very complex with numerous dependencies between different parts of feature hierarchies. These kinds of hierarchies quickly change into structures that lose all of their initial benefits. These and many more issues discussed in the work by Hein and colleagues have greatly influenced our work [16,17,27].

The method for requirements authoring and management (MRAM) proposed by Keepance and Mannion uses discriminants to structure requirements [20]. Discriminants describe how one product is different from another product variant. Single and option discriminants correspond to mandatory and optional features in the FODA method, whereas multiple discriminants are not mutually exclusive as the alternative features are. We share the definition of multiple discriminants in our variability tree model. The MRAM approach also supports deriving concrete product instantiations from the reference architecture by binding the variation points. However, their MRAM method does not address the dependency problems of the related variations points.

The software productivity consortium has proposed a method to perform domain modeling [1]. During domain analysis, they use commonality and variability assumptions to describe what is common between different systems and how the systems vary. A related method, the family-oriented abstraction, specification and translation (FAST) method relies on the same ideas of commonality and variability assumptions as the SRC method [2]. The FAST method focuses on developing a domain-specific language that implements the assumptions, which were discovered during domain analysis. Domain-specific languages (DSLs) can be used to specify variability and eventually create all needed family variants. The semantic structure of the DSL is engineered for this particular domain, and it therefore specifies how the product family can vary within the current scope by holding all relevant domain assumptions. A correctly implemented DSL would also include the dependencies between decisions and could ensure that these rules are

satisfied. Therefore, the created DSL includes all information on the relevant dependencies between features. However, using domain-specific languages hinders the ability to adapt and reengineer existing product families. In these environments, it is often impossible to propose completely new ways to handle variability and derive family variants. On the contrary, it is essential to carefully adapt the current way of working and use the existing design as the basis for reengineering. But the DSL can be created for a selected domain, as demonstrated by the FAST approach's application to the command and reports subsystem of the 5ESS telecommunications switch [32].

Viewpoint-oriented requirements management has become a commonly used paradigm among researchers [11,12,18,26]. Viewpoint-oriented methods concentrate on capturing separate descriptions that together form the complete specification. However, having these separate descriptions highlights the need for methods to identify and resolve possible overlaps and inconsistencies both within a viewpoint and across separate viewpoints. In our model, we preserve consistency by connecting dependency relation to the same features that are present in the feature tree. Checking the dependency view for any new features that don't exist in the feature tree occurs during the creation of the dependency model. If such features emerge, they must also be included in the feature model.

Many researchers have focused on studying the feature-interaction problem [24,25,33]. They have created methods to identify and resolve conflicts among features. The feature-interaction research targets studying the combined behavior features, whereas our work is interested mainly in the interactions as a tool to understand and configure the system better. Knowledge of the actual type of the dependency or interaction potentially helps us to refractor the system to remove those connections that prevent us from creating the optimal system structure.

6 Conclusion

We identified the main requirements of industrial-strength, feature-modeling methods. Then we explained why dependencies between features are a dominant problem in reengineering existing product lines. To support dependency analysis of existing product lines, we introduced a dependency analysis modeling technique and notation. We showed how our model can be applied in a concrete product family using two examples. Finally, we set the context by relating our work to other research about feature modeling.

The dependency modeling in the current feature-modeling methods belongs to two main categories. In the first category, there are methods that model dependencies between features using rules. Our method provides better tools to manage a large number of dependencies, and it supports the graphical modeling of key dependencies and thus improves the overall understanding of the system. The second category includes approaches that model dependency information directly to the feature trees that are normally used to represent variability. This approach has been validated to be impractical in large-scale use. Annotating feature diagrams with dependency information greatly decreases the usability of the feature models. By separating dependency information into another view greatly improved the understandability of the feature model.

The approach presented in this paper is based on our experience of feature modeling in multiple domains. In the future, we plan to create a feature dependency taxonomy that could be used in analyzing and communicating feature dependencies to all key stakeholders.

Acknowledgments

We would like to thank the engine-control domain experts Klaus Hirschmann, Magnus Labbé, and Elmar Pietsch for their patient support of our feature analysis.

References

1. Reuse-driven software processes guidebook. Technical Report Version 02.00.05, Software Productivity Consortium, November 1993.
2. M. Ardis and D. Weiss. Defining families: The commonality analysis. In *International Conference on Software Engineering ICSE1997*, pages 649–650. IEEE, 1997.
3. D. Batory, L. Goglianese, and M. Goodwin. Creating reference architectures: An example for avonics. In *Proceedings of the Symposium of Software Reuse*, pages 27–37. ACM, 1995.
4. J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Methig, K. Schmid, T. Widen, and J. DeBaud. PuLSE: A methodology to develop software product lines. In *Symposium on Software Reusability (SSR'99)*, pages 122–131, 1999.
5. Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. SEI Series in Software Engineering. Addison-Wesley, Reading, MA, 2001.
6. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, Reading, MA, 2001.
7. S. Cohen, J. Stanley, W. Peterson, and R. Krut. Application of feature-oriented domain analysis to the army movement control domain. Technical Report CMU/SEI-91-TR-028, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, June 1992.
8. K. Czarnecki and U. Eisenecker. *Generative Programming — Methods Tools and Applications*. Addison-Wesley, Reading, MA, 2000.
9. James C. Dager. Cummin's experience in developing a software product line architecture for real-time embedded diesel engine controls. In Patrick Donohoe, editor, *Software Product Lines: Experience and Research Directions*, pages 23–45. Kluwer Academic Publishers, Boston, 2000.
10. Alan Michael Davis. *Software requirements: Analysis & specification*. Prentice-Hall, Englewood Cliffs, NJ, 1990.
11. S. Easterbrook and B. Nuseibeh. Using viewpoints for inconsistency management. *BCS/IEE Software Engineering Journal*, pages 31–43, January 1996.
12. A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, pages 31–58, March 1992.
13. J. Gerhardt, H. Hönninger, and H. Bischof. A new approach to functional and software structure for engine management systems — BOSCH ME7. In *SAE International Congress and Exposition, Session: Electronic Engine Controls, Detroit, MI, (February, 1998)*, pages 1–12, Warrendale, PA, 1998. Society of Automotive Engineers (SAE).
14. Jürgen Gerhardt. ME-motronic engine management. In *Gasoline-engine management*, pages 306–359. Robert Bosch GmbH, Stuttgart, Germany, 1999.
15. M. Griss, J. Favaro, and M d'Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of Fifth International Conference on Software Reuse*, pages 76–85. IEEE, 1998.

16. A. Hein, J. MacGregor, and S. Thiel. Configuring software product line features. In *Workshop on Feature Interaction in Composed Systems 15th European Conference on Object-Oriented Programming (ECOOP 2001), Budapest, Hungary (June 18–22, 2001)*, 2001.
17. A. Hein, M. Schlick, and R. Vinga-Martins. Applying feature models in industrial setting. In P. Donohoe, editor, *Software Product Lines — Experience and Research Directions*, pages 47–70. Kluwer Academic Publishers, Boston, 2000.
18. A. Hunter and B. Nuseibeh. Managing inconsistent specifications: Reasoning analysis and action. *ACM Transactions on Software Engineering and Methodology*, pages 335–367, October 1998.
19. K. Kang, S. Cohen, J. Hess, W. Novak, and S Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.
20. B. Keepance and M. Mannion. Using patterns to model variability in product families. *IEEE Software*, pages 102–108, July/August 1999.
21. M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attribute-based architecture styles. In Patrick Donohoe, editor, *Software Architecture*, volume 140 of *International Federation for Information Processing*, pages 225–243. Kluwer Academic Publishers, Boston, 1999.
22. J. Kuusela and J. Savolainen. Requirements engineering for product lines. In *International Conference on Software Engineering (ICSE2000)*, pages 61–69, 2000.
23. K. Lee, K. Kang, W. Chae, and B. Choi. Feature-based appraoch to object-oriented engineering of applications for reuse. *Software Practise and Experience*, pages 1025–1046, 30 2000.
24. Louise Lorentsen, Antti-Pekka Tuovinen, and Jianli Xu. Experiences in modelling feature interactions with coloured petri nets. In Tibor Gyimothy, editor, *Proceedings of Seventh Symposium on Programming Languages and Software Tools SPLST 2001*, pages 221–230. University of Szeged, 2001.
25. Louise Lorentsen, Antti-Pekka Tuovinen, and Jianli Xu. Modelling feature interaction patterns in nokia mobile phones using coloured petri nets and Design/CPN. In *Proceedings of Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, CPN'01, Aarhus, Denmark, August 29-3*, 2001.
26. N. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirement specification. *IEEE Transactions on Software Engineering*, pages 760–773, October 1994.
27. Michael Schlick and Andreas Hein. Knowledge engineering in software product lines. In *European Conference on Artificial Intelligence (ECAI 2000), Workshop on Knowledge-Based Systems for Model-Based Engineering, Berlin, Germany (August 22nd, 2000)*, 2000.
28. STARS. Organization domain modeling (ODM) guidebook, version 2.0. Technical Report STARS-VC-A025/001/00, Software Technology for Adaptable, Reliable Systems (STARS), June 1996.
29. Steffen Thiel, Stefan Ferber, Thomas Fischer, Andreas Hein, and Michael Schlick. A case study in applying a product line approach for car periphery supervision systems. In *In-Vehicle Software 2001, SAE 2001 World Congress, March 5-8, 2001, Cobo Center, Detroit, Michigan*, volume SP-1587, pages 43–55, Warrendale, PA, 2001. Society of Automotive Engineers (SAE).
30. A. D. Vici, N. Argentieri, A. Mansour, M. d'Alessandro, and J. Favaro. FODAcom: An experience with domain analysis in the italian telecom industry. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 166–175, Los Alamitos, CA, 1998. IEEE Computer Society Press.

31. Jens Weber and Karlheinz Topp. Information technology — a challenge for automotive electronics. In *Embedded Software Session (PC34), SAE 2001 World Congress, March 5-8, 2001, Cobo Center, Detroit, Michigan*, number 2001-01-0029, Warrendale, PA, 2001. Society of Automotive Engineers (SAE).
32. D. Weiss and R. Lai. *Software Product-Line Engineering — A Family-Based Software Development Process*. Addison-Wesley, Reading, MA, 1999.
33. P. Zave. Feature interactions and formal specification in telecommunications. *IEEE Computer*, pages 20–29, August 1993.