Modelling Dependencies between Variation Points in Use Case Diagrams

Stan Bühne, Günter Halmans, Klaus Pohl
Institute for Computer Science and Business Information Systems (ICB)
Software Systems Engineering
University of Duisburg-Essen
45117 Essen; Germany
E-mail: {buehne, halmans, pohl}@sse.uni-essen.de

Abstract

Software product family variability facilitates the constructive and pro-active reuse of assets during the development of software applications. The variability is typically represented by variation points, the variants and their interdependencies. Those variation points and their variants have to be considered when defining the requirements for the applications of the software product family. To facilitate the communication of the variability to the customer, an extension to UML-use case diagrams has been proposed in [9].
In this paper we identify common dependency types used within the feature modelling community to express interdependencies between variation points and variants. We then propose a differentiation of those interdependencies to be used to reduce complexity and to facilitate selective retrieval of interdependencies between variation points and the variants. Finally, we extend the notations proposed in [9] for representing interdependencies between variation points and variants in use case diagrams and use a simple example to illustrate those extensions.

1 Introduction

Software product families facilitate pro-active and constructive reuse of software product assets and thereby reduce the development costs of customer applications. The development of product families is characterized by two processes (cf. Figure 1). During domain engineering the common assets are realized and the variability of the product family is defined. During application engineering the product family variability is exploited to define and implement different products by reusing the shared assets defined during domain engineering [21].

Obviously, making the costumer aware of the product family capabilities and variability is a key factor for the successful reuse of the product family assets. The communication of the product family variability to the customer is thus essential. Communicating the capability and the variability to the customer is the main difference between requirements engineering for single software products and requirements engineering during domain engineering.

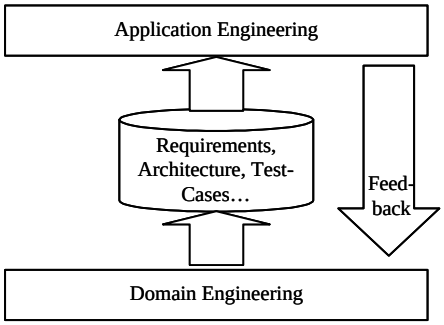

Figure 1: Domain and Application Engineering in Product Family Development

Most research contributions on product family variability focus on the architectural level [1], [2], [20]. For example, they deal with aspects like the realization of variability on a technical level and technical binding of the variation points. From the customer perspective those aspects of variability are not so important. In contrast, the customer is rather interested in the essential aspects of the variability, i.e. the customer wants to know how the variability contributes to his needs. Thus the customer is e.g. interested in the functional and quality aspects of variability, i.e. which different payment methods are available for an internet-based booking system. As argued in [9] one should thus distinguish between *essential* and *technical* variability.

Essential variability comprises all types of variability from the customer viewpoint. This includes of course all functional aspects. For example, the provider of a navigation system provides two variants for determining the address of the next destination, via a menu or via voice input. The customer would be interested in having the choice between the different “input” methods, whereas the engineer is more interested in implementation aspects, e.g. how the voice input is technically integrated in the navigation systems.

Obviously, when defining the requirements for a customer specific application during application engineering, at least the essential variability aspects have to be communicated to the customer. In other words, the variation points and variants have to be considered when defining the requirements for the applications of the software product family. To facilitate the communication of the essential variability aspects to the customer we proposed an extension to UML-use case diagrams (cf. [9]).

Adding to the complexity, besides the variants and the variation points also the interdependencies between variants and variation points have to be considered. For example, a variant can exclude another one, e.g. when you are buying a roof-less car, you cannot choose a sun-roof. Or, a variant might require the choice of another variant, e.g. if you choose an engine with high power, you have to take bigger wheels. Obviously, only if the customer is aware of those interdependencies he is able to recognize the consequences of his selection. Since there are generally many of such interdependencies within a product family, dealing with the complexity of those interrelations is a big challenge.

As the major step forward regarding our work in [9] in this paper we focus on how those interdependencies between variants and variation points can be communicated to the customers. We propose a classification as a mean to deal with the complexity of the interdependencies and suggest a representation using common requirements engineering techniques for supporting the communication.

In section 2 we briefly describe the extensions to use case diagrams required for representing variation points and variants.

In section 3 we give an overview on the state of the art of considering interdependencies of product family variability at the requirement level.

To be able to deal with the complexity of those interdependencies we suggest to consider the origin of the dependencies and to clearly indicate derived interdependencies in section 4.

In section 5 we outline how those interdependencies can be visualized in UML Use Case Diagrams and sketch potential tool support.

In section 6 we summarize the main contribution of this paper and provide an outlook on future work.

2 Representing Product Family Variability in Use Case Diagrams

To facilitate the communication of variation points and their variants to the user, both the variation points and the variations must be explicitly represented. In [9] we suggested to apply Use Cases for communicating the variability of the product family to the customer and showed why standard UML notations are not suitable for this purpose. Therefore, we proposed extensions to Use Case Diagrams, which enables the explicit representation of variation points and their variants (see [9] for details). In the following we briefly introduce these extensions.

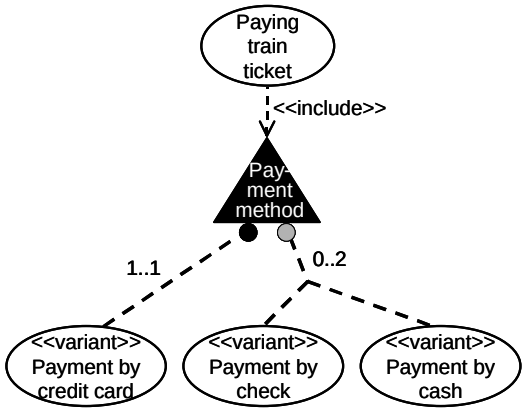

Figure 2: Explicit Representation of Variation Points in Use Case Diagrams

- ***Variation point***: For making a variation point visible in use case diagrams we represent a variation point as a triangle. The variation point is itself included by another use case (cf. Figure 2)
- ***Variant***: To make variant use cases explicit we use the stereotype <<variant>> (cf. Figure 2, e.g. the variant *payment by credit card*)
- ***Cardinality of the relationship between variation point and variant***: Regarding the relationship between variants and variation points one has to consider, that a variant can be mandatory for a variation point A and optional for a variation point B. To express these kinds of relationships we represent the relationship between variants and variation points by a dashed line and a circle on the variation point side. Moreover, we define cardinality for the relationship to express if a variant is mandatory or optional regarding the specific variation point. More general the cardinality expresses, how much of the related variants *must* and how much of the related variants *could* be selected. In Figure 2 for example the variant *payment by credit card* is mandatory, which is represented by the cardinality 1..1.
- ***Mandatory and optional variation points***: To make clear if one of the related variants according to a specific variation point has to be chosen we distinguish between mandatory and optional variation points. A variation point is mandatory, if at least one of the related variants has to be selected. Mandatory variation points are represented by a black filled

triangle (like the one in Figure 2) whereas optional variation points are represented by a light-grey triangle.

With the defined extensions variation points and variants are explicitly visible in the use case diagram. Furthermore the cardinality of the relationship between variant and variation point is documented. It is now explicit, whether a variation point is mandatory or not and one is able to recognize how much variants from a set of variants have to be selected.

The new notations do not involve the possibility to represent dependencies, e.g. between variants related to another variation point. Furthermore, so far it is not possible to express dependencies between variation points and to represent the type of dependencies. These issues will be our focus in the next sections.

3 Dependencies in Feature Modelling

The need to consider the interdependencies between different variation points at the requirements level has been identified already by the feature modelling community. In this section, we provide an overview on the state of the art of representing dependencies between features. In Table 1 we first give an overview of dependency-types used in feature modelling approaches.

Dependency Type	**Description**	**Used in**
composed of	The composed of dependency is used when a parent feature is consisting of a set of child features (e.g. mobile is composed of voice-transfer, data-transfer, shot messages, etc.).	FORM [15] FOPLE [16]
implemented by	The implemented by dependency is used to describe that one feature is needed to implement another feature (e.g. UMTS needs specific transfer protocols).	FORM [15] FOPLE [16]
generalization	The generalization / specialization dependency is used to generalize or specialize features (e.g. data transfer can be specialized as wap, UMTS, fax, etc.).	FORM [15] FOPLE [16]
refinement	The refinement link is used to structure features in the same way as the composed of dependency	FODA [14] FeatuRESB Riebisch
requires	The requires dependency is used to describe if one feature needs another (e.g. satellite-navigation requires GSP-signal)	FODA FeatuRESB [12] Riebisch [19]
(mutual) exclusive	The exclusive (or exclude) dependency is used when one feature conflicts with another (e.g. the textual cellular display excludes mobile photography)	FODA [14] FeatuRSEB [12] Riebisch [19]
hints	The hint dependency is a strategic dependency, and is used to express that the choice of another feature increases the system usage (e.g. mp3 player for mobile)	Riebisch [19]
mathematical	The mathematical dependency describes the relative impact from one feature to another.	Riebisch [19]

Table 1: Dependency-Types used in Feature Modelling

FODA [14] was one of the first feature-oriented approaches. For the representation of dependencies FODA uses refinement-links to partition commonalities and variable aspects among features and semantically describes constraints for requires and mutual exclusive dependencies. FOPLE [16] employs composition, generalization, and implementation links to structure variable aspects among features. The FORM – Method [15] supports the same kind of links as the FOPLE approach. The FeatuRSEB approach [8] is a combination of FODA and Jacobsons RSEB [12], and therefore uses the same dependencies as in FODA to describe variability. Riebisch et al. [19] use refine links to express variability aspects and further differentiate between hard constraints, as requires and mutual exclusive, and soft constraints as hints and mathematical relations to describe dependencies between features.

The various dependency types proposed in the feature modelling literature can be characterized by two main categories:

- *Realization-dependencies*: This category subsumes all dependencies, which deal with realization aspects of a feature. Such dependency types are used to define the restriction in the choice of the variants associated to one variation point. E.g. the representation of a

navigation-system can be by voice, graphics, and or text whereas the representation is either colored or black and white. These dependencies show in which different ways an aspect can be realized.

- *Constraint-dependencies*: This category subsumes all other feature interdependencies. For example, one feature can exclude another, or one feature requires another feature like the choice of colored-graphical representation within a navigation-system requires a colour display and geographical information to fulfil this demand.

4 Considering the "Why" of Product Family Variability Interdependencies

A prerequisite to find a suitable representation for product family variability dependencies is to know why these dependencies exist.

In this section we elaborate on different reasons (sources), which cause a dependency concerning the product family variability. Based on dependency types used in feature modelling (see last section) we first define a set of dependency types to be used to represent interdependencies between variation points and variants (Section 4.1). In Section 4.2 we discuss the various types of possible interdependencies between variation points and/or variations. In Section 4.3 we elaborate on the reasons why a certain type of dependency is introduced. We thereby distinguish between so-called core-dependencies and derived-dependencies. In Section 0 we differentiate between several types of derived-dependencies.

4.1 Dependency types

Based on the dependency types used in feature modelling and our experience with modelling dependencies concerning product family variability, we suggest differentiating at least between the following four types of interdependencies:

- *requires-dependency*: describes that the binding of one variant implies the need of another variant (required variant). For example if one wants to lock up his car via remote control this variant requires a centralized door locking.
- *exclusive-dependency*: describes that the binding of one variant excludes the selection of another variant (excluded variant) – means that only one of those variants can be selected. As mentioned earlier the choice to get a cabriole for example excludes the variant "sunroof".
- *hints-dependency*: In analogy to the approach of Riebisch et al. [19] this dependency type comprehends dependencies where the binding of one variant has some positive influence on another variant. For example a T-DSL-connection might have a positive influence on choosing the online shopping variant (in contrast to a modem-connection).
- *hinders-dependency:* describes that the binding of one variant has some negative influence on another variant. If one chooses the mobile phone for using the internet this might have a negative influence on online shopping because of the little mobile phone displays.

Note, that the requires dependencies and the hint dependencies are uni-directional, whereas the exclusive-dependency and the hinders dependencies are bi-directional. For example, if a variant A requires a variant B it does not imply that the variant B requires the variant A. In contrast, an exclusive-dependency between variant A and variant B represents that if variant A is chosen than variant B can not be chosen and vice versa.

Dependency type	**Direction**
Requires-dependency	Uni-directional
Exclusive-dependency	Bi-directional
Hint-dependency	Uni-directional
Hinders-dependency	Bi-directional

Table 2: The Direction of Dependency Types

4.2 Dependencies between Variants and/or Variation Points

The following three principle interrelations between variants and/or variation point exists:

- *Dependency between variant and variation point.* As described in [2] and [9] a variant is obviously related to one or more variation points. For example the variant *paying a ticket by credit card* is a variant of the variation point *payment methods for train tickets* (in addition to the variants *payment by check and payment by cash*). A variant can be related to more than one variation points (and has thus more than one dependencies to a variation point). The variant *paying a ticket by credit card* for example could also be a variant of the variation point *payment method for merchandising articles* (of the train company).
- *Dependency between variant and variant*: Consider the variant *paying by credit card.* If this variant will be selected for paying via internet one has also to select the specific variant for encrypting the credit card information. In this category one has to consider two different cases: (1) the depending variants are related to one variation point and (2) the depending variants are related to different variation points (see the example mentioned above).
- *Dependency between variation point and variation point*: For example, if a mobile phone product family provides the choice to use the mobile phone in different countries (variation point *countries*) it also

has to provide the choice between different protocols (variation point *protocol*).

The dependencies described in this Section are orthogonal to the dependency types in Section 4.1. For example, the dependencies between a variant and a variation point can be a require dependency or an exclusive dependency and a require dependency can occur in all three categories, i.e. between variation points, between variants and between a variant and a variation point.

4.3 Core Dependencies

Each dependency is of course introduced for a particular reason. For being able to understand the reason why a dependency was introduced in the first place, we suggest to add a textual attribute to each dependency for describing the reasons.

We thereby distinguish between two types of dependencies with respect to the product family variability:

- *core-dependencies*, which subsume all dependencies introduced due to the context of the system, i.e. the system context enforces some constraints on the system represented as core-dependencies.
- *Derived-dependencies*, which subsume all dependencies derived from the core-dependencies. One reason why a derived-dependency exists lies in the refinement of features and functions. But there are others (see section 4.3).

For the core-dependencies we further suggest to classify the influence the context of the systems has on the dependencies existing between features (variation points, variants) according to three main views (see Figure 3; cf. [11], [18] for a detail description of those views/worlds):

- *Usage view*: from the usage point of view the system under consideration has to support needs respectively solve problems of the users (whereby a user can be the end-user, the provider or administrator of the system). Thus the usage view deals with all functional demands to the system – in other words it deals with the whole functionality the system should provide to any user.
- *Domain view*: from the domain point of view the system under consideration needs to map the corresponding real world domain to a system. That means all necessary information of a specific scope (domain) has to be represented in a system. This data is driven by relevant subject properties (e.g. name, address, etc of a vendee), domain specific standards (e.g. financial standing) or driven by laws (e.g. full name and address of vendor).

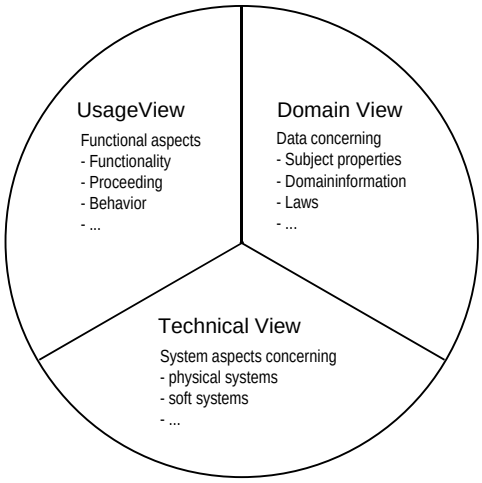

Figure 3: Contextual Views of a System

- *System/technical view*: from the technical point of view the system under consideration depends on physical systems, means given and requested hardware (e.g. architecture, communication interfaces) and soft systems means given and requested software (e.g. computer- and user-interfaces). The term physical systems enfold all environmental systems the product has to deal with (e.g. signal of GPS-Satellite), as well as physical restrictions (e.g. size of display, number of buttons, etc.).

Those views are obviously interrelated, e.g. facts about the domain are represented by systems to support the end-user with needed functionality.

Source	**Category**	**Example**
Usage View	Dependency regarding the functionality of a system	the payment method via credit card and internet depends on the security functionality of the web application
Domain View	Dependency regarding the data of the system	the choice to represent a signature may be depend on the representation of additional authentication information
Technical View	Dependency regarding the IT of a system	a chosen graphical user interface (e.g. mobile phone) affects the needed implementation language (e.g. WAP).

Table 3: Dependency Categories of Core Dependencies

Table 3 summarizes the three different contextual sources for a dependency: the domain view focuses on the data (information), the usage view focuses on functional aspects of the system, and the technical view focuses on computer systems, networks, telecommunication systems, etc. the system run on or interacts with.

4.4 Derived Dependencies

Many dependencies concerning the product family variability are not introduced due to contextual reasons, i.e. there are many dependencies, which are not core-dependencies, but derived from the core dependencies or from so-called high-level features.

4.4.1 Dependencies Derived from High-Level-Features

A main reason for introducing variation points and/or variants can be the existence of a so-called high-level features. Examples for such high-level features are "sports-edition" in the car industry or "internet-shop" in a procurement system. For example, when buying a car you might be able to select the "sports-edition". By making this decision, you select a lot of "details", i.e. you bind a whole set of variation points with a predefined set of variants defined for this high-level feature. For example, you get a strong engine, an aluminium interior (instruments, knops, etc.), special tires etc.

Thus, a high level feature often predefines the "binding" of a whole set of variation points to specific variants. Moreover, a certain variant or even a whole variation point might only exist due to the high-level feature.

We thus suggest to specify explicitly that a variation point and/or a variant were introduced due to a high level feature. This can be achieved by adding a dependency link stating the high-level feature was responsible for the introduction of the variant or variation point and by adding an attribute to each dependency link between variants/variation points introduced due to high-level feature.

4.4.2 Transitive -Dependencies

Transitive dependencies are derived from other dependencies. A functionality required from the usage perspective (usage view) might require specific information/data, which in turn require some it-solution. For example, to support the evaluation of the actual stock market situation, one requires the actual stock exchange rates which in turn require a IT-solution providing an update of the stock exchange data every 10 seconds. Figure 4 illustrates this situation. There are two core-dependencies (between the functionality and the data, and between the data and the IT) and one indirect dependency, specifically between the functionality and the IT. The latter one we call (derived) *transitive-dependency*.

To make the reasons traceable, we argue that the derivation of transitive-dependencies should be visible to the product family engineer and the customer. We thus indicate the fact that a dependency was derived due to other core-dependencies by a "transitive-dependency" attribute, which relates the derived dependency to the core-dependencies it was derived from.

Knowing where a dependency was derived from (e.g. to retrieve the sources for the derivation) enables an engineer to understand why a specific dependency exists. This information facilitates the selection of the variants during application engineering, and – even more important – is very helpful for dealing with evolution of the product family itself (e.g. when adapting a specific variant for a customer or even changing a variant for the whole product family).

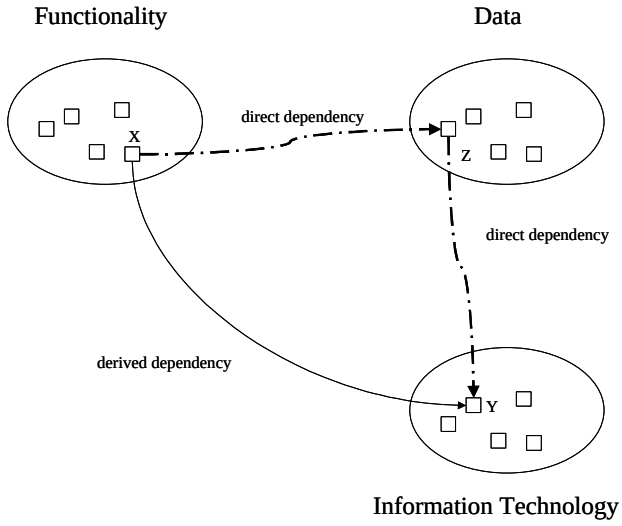

Figure 4: Derived Transitive Dependencies

4.5 Classification of Dependencies: Summary

Differentiating between derived and core dependencies empowers the user to focus on a special type of dependencies at a time and thus decreases the complexity of dependencies within product family variability significantly (see section 5 for examples). This differentiation is thus essential for facilitating the communication of product family dependencies to customers. In Figure 5 the necessary attributes for the dependency-links are depicted.

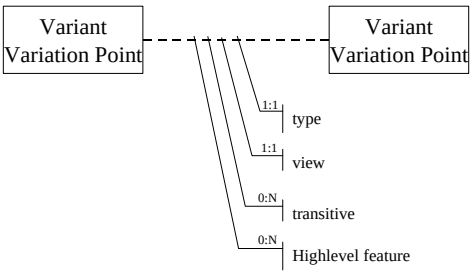

Figure 5: Dependency Attributes

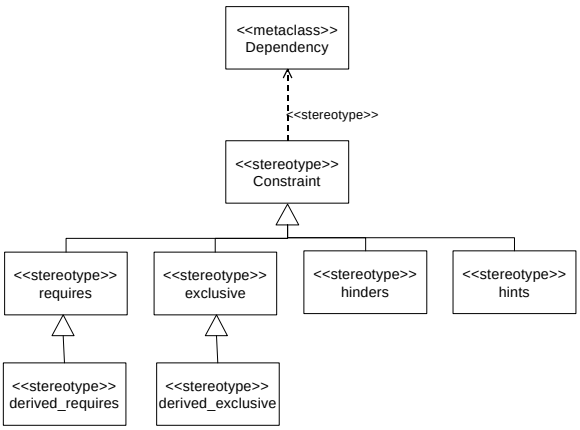

Figure 6: Dependency-Stereotypes

5 Representation of Dependencies in Use Case Diagrams

In Section 4 we described the different sources of dependencies within product family variability. In this section we suggest a representation of the different dependencies to support the communication to customers. Motivated by the positive experiences with the representation of variability aspects by use case diagrams (see [9]) we suggest completing this representation according to dependency aspects. The intention of use case diagrams is to model the interaction of users and/or other systems with the system under consideration. Therefore, in the following section we concentrate on the functional aspects according to the product family dependencies.

5.1 Representing the Four Types of Dependencies

To express dependencies in use case diagrams, we need to extend the notation among the mentioned dependency-types. The UML [17] allows extending the language by refining consisting elements by stereotyping. To express the dependencies we created a <<requires>>, <<exclusive>>, <<hints>>, and <<hinders>> stereotype. To express the derived dependencies we also specialized the requires and exclusive dependency type to <<derived_requires>> and <<derived_exclusive>> (see Figure 6).

The <<requires>> type is expressed by a dotted line with an arc to the element that is required by the other. The <<exclusive>> type is expressed by a dotted line with arcs on each end that illustrate the exclusive relation between these elements. The <<hints>> type is expressed by a pointed line with an arc to the useful element. The <<hinders>> type is illustrated by a pointed line with an arc on each end, that illustrates the hindering relation between those elements. (Figure 7)

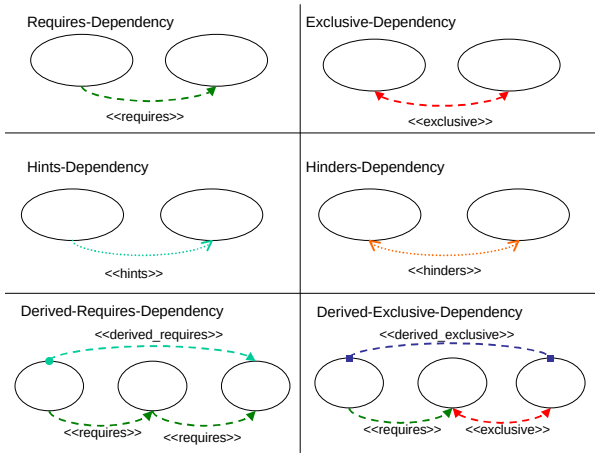

Figure 7: Dependency-Types

5.2 Representing the Dependencies According Variant and/or Variation Points

Dependencies between Variants of the same Variation Point

Dependencies among variants of the same variation point, like the selection of one variant requires another, have to be expressed by dependency-links. For example,

the selection of the variant *viewpoints* <<hints>> the selection of *city maps* (Figure 8).

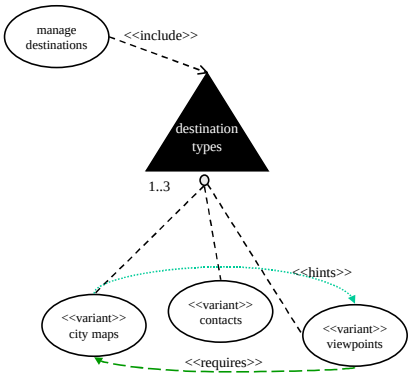

Figure 8: Dependencies between Variants of one Variation Point

Beside the represented requires- and hints-dependency, also exclusive- and hinders-dependencies have to be represented.

Dependencies between Variants of different Variation Points

As already mentioned before, dependencies between variants of different variation points also have to be handled. E.g. the selection of a specific variant *"route selection by viewpoint"* causes a requires-dependency to the variant *"manage viewpoints"* of another variation point (Figure 9).

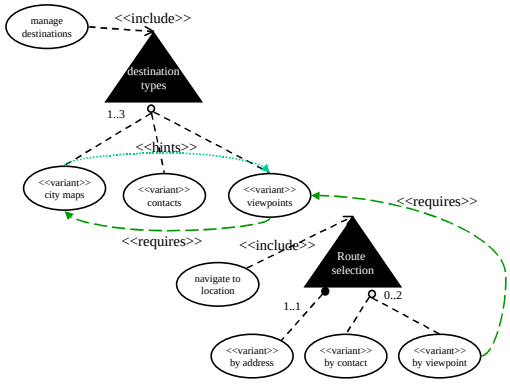

Figure 9: Dependencies between Variants of different Variation Points

Dependencies between Variation Point and Variation Point

The selection of one variation point can cause dependencies to other variation points. For example the selection of a movie-player requires the selection of an audio-player. In Figure 10 we illustrate an example where the selection of a navigation system *"navigate to location"* requires the use case variation point *"manage destinations"*.

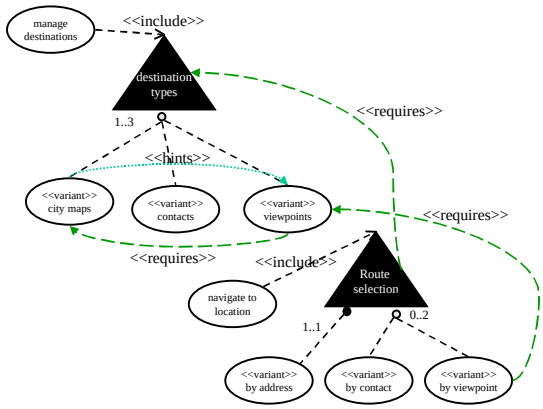

Figure 10: Dependencies between Variation Points

Dependencies between Variants and Variation Points

The selection of a variant can cause dependencies to other variation points (without a restriction to the selected variants). This means that when selecting a variant another variation point is required or even excluded, without consideration of the variants. Figure 11 illustrates this case: the variant "*manage contacts*" requires the variation point "*admin categories*", without specifying any variants of the required variation point.

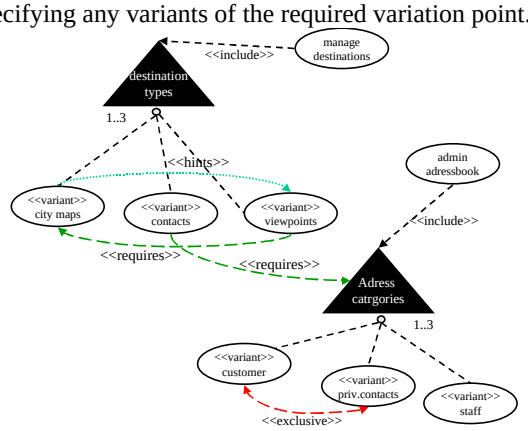

Figure 11: Dependencies between Variants and Variation Points

5.3 Representing the Core of Dependencies

In 4.3 we introduced origins of dependencies and differentiated among three categories: usage, domain, and technology. The example (Figure 12) shows the origin of the requires-dependency between "*viewpoints*" and "*city maps*". The depicted requires-dependency has its origin in the domain, because to manage the viewpoints of a city, one needs city maps to locate those viewpoints.

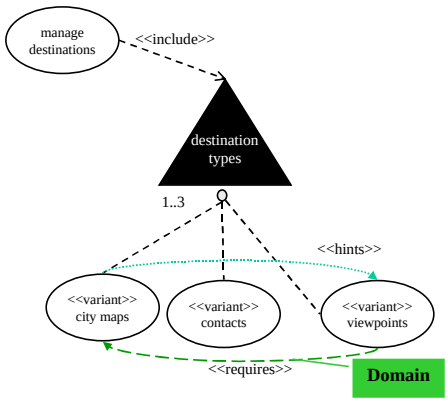

Figure 12: Dependencies with Core Description

The origins of the dependency-links will be illustrated as additional attribute to the link-type. By an adequate tool-support one can generate views that express which use case dependencies have their origin in the domain (i.e. in data), in information technology, or in the usability of a system. As a result of that, one has the opportunity to discover which use cases change when for example the domain, or information technology changes.

5.4 Representing derived dependencies

5.4.1 Derived Dependencies form High Level Selections

When selecting a high level feature the binding of a whole set of variants or variation points can be predefined and/or the user has to make a selection for several variation points. For example, if the user selects for a shop system the variant *online system*, he can choose between payment and transaction methods, and delivery different delivery services. But if he selects the *local store* variant, he is not able to choose different transaction methods. The selection he can make therefore depends significantly on the choice he made on the higher level. The example in Figure 13 shows that as a result of a high level selection ("online shop") new (derived) dependencies appear that have to be considered.

In the example, both requires-dependencies are the reason for the "new" derived dependency.

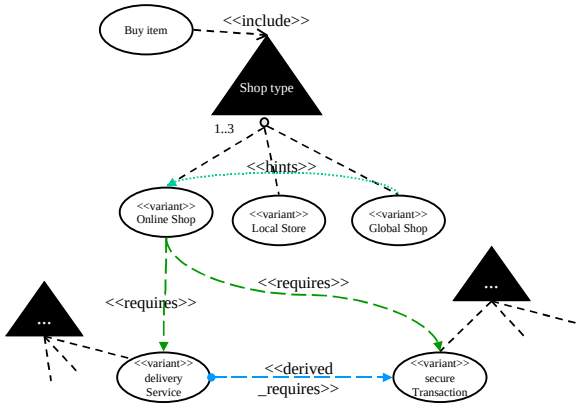

Figure 13: Derived Dependencies Evoked by High Level Selections

5.4.2 Transitive Dependencies

We discussed the fact that a dependency is transitive (derived from other dependencies) in 4.4.2. As depicted in Figure 14, UC_1 requires UC_2 and UC_2 requires UC_3. Therefore it is obvious that UC_1 indirect requires UC_3. The representation of derived dependencies in use case diagrams generates a huge complexity of those diagrams. Figure Figure 14 illustrates the complexity on a simple but abstract example..

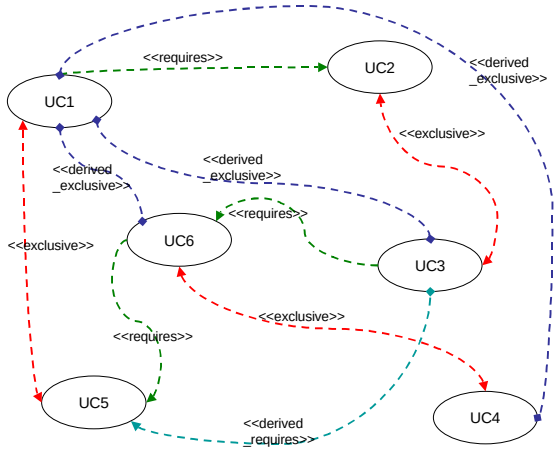

Figure 14: Derived Transitive Dependencies

Figure 15 illustrates the overall complexity of dependencies among use case variants and variation points. Even if only three variation points are illustrated as in this example the result is quite complex. When we think of product families one has to deal with hundreds of variation points. To handle this complexity assistance by tools is necessary.

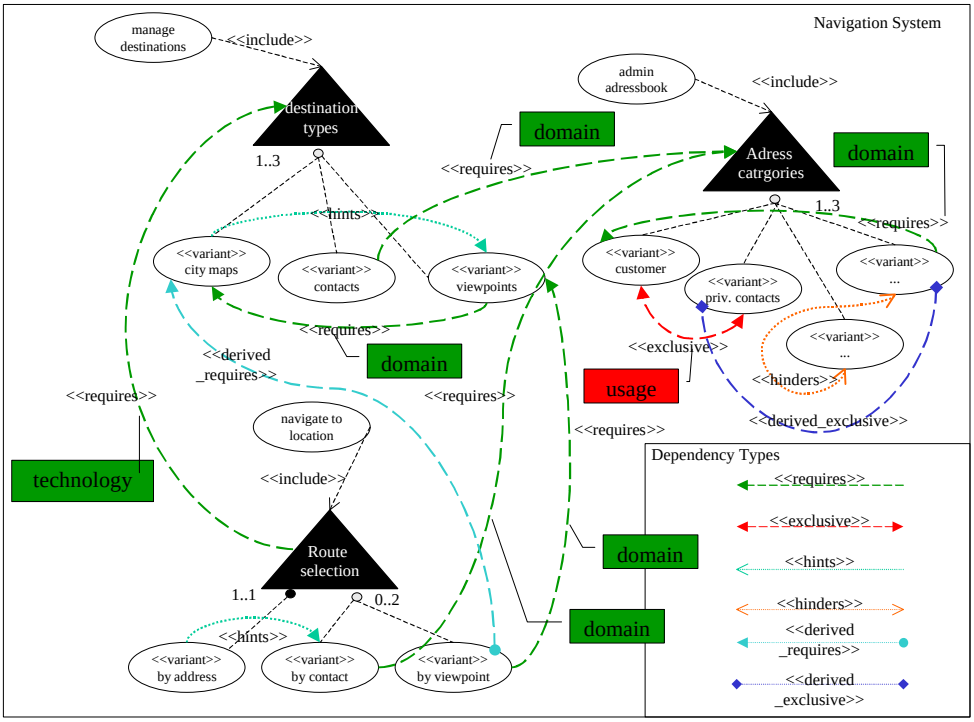

Figure 15: Overall Dependency View

5.5 Tool Support

Considering the representations of the different dependencies (sources and types) one has to recognize the enormous complexity of product family variability dependencies. Thus for the communication of these dependencies to customers it is essential to reduce this complexity. Thus the communication to customers leads to the following two problems:

- On the one hand, one has to avoid that the use case diagrams are overloaded when representing the interdependencies.
- On the other hand, the customer has to recognize the important ones. Only if he is aware of the important dependencies he will be able to recognize the consequences of his selections.

The problems can be solved by tool support and by using the different categories of dependencies outlined in Section 4. The tool should fulfil the following requirements:

- In the standard mode the tool only should visualize the four dependency types described in Section 4.1. The tool should provide a separate view for each dependency type, so that it is possible for example to see all exclude-dependencies within one view. In this standard mode no sources of core dependencies (usage view, domain view, IT view) or derived dependencies are shown. (5.2)
- Then the tool should be able to visualize the different sources of dependencies so that the customer is able to recognize if a dependency is based on functional or data aspects. (5.3)
- Furthermore the tool must be able to visualize the derived dependencies, from high-level selections and transitive dependencies (5.4). This implies to
  - o show the dependencies (core and derived) in a graphic form. It must be possible to fade in and fade out the derived dependencies.
  - o highlight the origin of derived dependencies
  - o represent the derived dependencies in a tabular form

With these capabilities the tool would be able to support the communication of product family variability dependencies. One would be able to generate the specific view according to the current topic of the product definition. The tool thus is able to reduce complexity and to make the customer aware about the essential dependencies.

6 Summary and Outlook

One of the main concepts of product families is to facilitate the constructive reuse of product family core assets during the development of customer specific products. The main goal of reusing core assets is to reduce costs and time in the process of developing

products. In order to reach this goal it is essential to communicate the product family variability and capabilities with the customer.

In this paper we argue, that explicit representation of product family variability and the variation point dependencies with use case models help the requirements engineer to communicate variability aspects to the customer. For finding suitable representations of dependencies we elaborated on different types of dependencies, in analogy to the dependencies within feature modelling. We characterized the dependencies between variants and/or variation points. Then we described why dependencies occur: The origin of dependencies can be caused by functional, data, or IT aspects. Moreover, dependencies can be derived from other dependencies and thus they are very difficult to recognize and to communicate.

Regarding the special interest of customers in functional aspects of variability and motivated by the successful representation of functional variability in use case diagrams we suggested a representation of dependencies in use case diagrams. The explicit dependency representation is a prerequisite (in addition to variant and variation point representation) of a successful communication of product family capabilities to customers. In addition it supports change management of variants because now one is able to identify all depending variants (or variation points) of the changed variant.

But the suggested representation of dependencies in use case diagrams also illustrates the enormous complexity of this issue. Therefore we suggest tool-support for reducing complexity without loosing the essential dependency information for the communication to the customers.

However, according to the definition and implementation of customer specific products many open issues have to be handled. Thus our future work deals with the following questions:

- How to represent non-functional aspects of variability, including dependencies?
- How to support the derivation of products using use cases and scenarios?
- How influences the cardinality of relationships between variants and variation points the dependencies between variation points?

7 References

[1] Felix Bachmann, Len Bass: Managing Variability in Software Architecture; ACM Press, NY, USA, 2001

[2] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, Henk Obbink, Klaus Pohl: Variability Issues in Software Product Lines; Fourth International Workshop on Product Family Engineering (PFE-4), Bilbao, Spain, 2001

[3] K. Czarnecki, U. Eisenecker; Generative Programming: Methods, Techniques, and Applications; Addison-Wesley 1999

[4] J. Carroll: The Scenario Perspective on System Development, Scenario-Based Design: Envisioning Work and Technology in System Development; John Wiley & Sons, 1995

[5] Paul Clements, Linda Northrop: Software Product Lines, Practices and Patterns; SEI Series in Software Engineering; Addison Wesley, 2001

[6] Alistair Cockburn: Writing Effective Use Cases; Addison Wesley, 2001

[7] Coriat, M., Jourdan, J., Boisbourdin, F. : The SPLIT method. Proceedings of the First International Software Product-Line Conference (SPLC-1), August 2000

[8] M. Griss, J. Favaro, M. d'Alessandro: Integrating Feature Modeling with the RSEB; 5th ICSR Conference Proceedings, Vancouver, Canada 1998

[9] Günter Halmans, Klaus Pohl: "Communicating the Variability of a Software Product Family to Customers", *Software and Systems Modeling*; Vol. 2, 15-36; Springer; Hamburg; March 2003;

[10] Günter Halmans, Klaus Pohl: Modellierung der Variabilität einer Produktfamilie; *Modellierung 2002: Modellierung in der Praxis - Modellierung für die Praxis*, 2002

[11] Matthias Jarke, Klaus Pohl; Establishing Visions in Context: Towards a Model of Requirements Processes (1993) EMISA Forum

[12] I. Jacobson, M. Griss, P. Jonsson : Software Reuse Architecture, Process and Organization for Business Success; Addison-Wesley:Longman, May 1997

[13] Ivar Jacobson: Object-Oriented Software Engineering: A Use Case Driven Approach; Addison Wesley, 1992

[14] K. Kang et al; Feature-Oriented Domain Analysis (FODA) Feasibility Study; Technical Report No. CMU/SEI-90-TR-2, November 1990, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA

[15] K. Kang et al; FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures; Annals of Software Engineering 5, 1998

[16] Kyo C. Kang, Kwanwoo Lee, JaeJoon Lee, and SaJoong Kim: Feature-Oriented Product Line Software Engineering: Principles and Guidelines; to in Domain Oriented Systems Development-Practices and Perspectives, Gordon Breach Science Publishers, UK, 2002

[17] Object Management Group (OMG); OMG Unified Modelling Language Specification, Version 1.4; September 2001 OMG, http://www.omg.org/uml

[18] Klaus Pohl; Process-Centered Requirements Engineering; Research Studies Press 1996

[19] Detlef Streiferdt, Matthias Riebisch, Ilka Philippow; Formal Details of Relations in Feature Models; In: Proceedings 10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems, Huntsville Alabama, USA, April 7-11, 2003

[20] Mikael Svahnberg, Jilles van Gurp, Jan Bosch: On the Notion of Variability in Software Product Lines; Proceedings of The Working IEEE/IFIP Conference on Software Architecture, 2001

[21] Frank van der Linden: Software Product Families in Europe: The Esaps & Café Projects; IEEE Software, 19(4); 41-49, July/August 2002