COVAMOF: A Framework for Modeling Variability in Software Product Families

Marco Sinnema, Sybren Deelstra, Jos Nijhuis, Jan Bosch

Department of Mathematics and Computing Science, University of Groningen,
PO Box 800, 9700 AV Groningen, The Netherlands,
{m.sinnema | s.deelstra | j.a.g.nijhuis | j.bosch}@cs.rug.nl, http://segroup.cs.rug.nl

Abstract. A key aspect of variability management in software product families is the explicit representation of the variability. Experiences at several industrial software development companies have shown that a software variability model (1) should uniformly represent variation points as first class entities in all abstraction layers (ranging from features to code) and (2) allow for hierarchical organization of the variability. It furthermore (3) should allow for first class representation of simple, i.e. one-to-one, and complex, i.e. n-to-m, dependencies, and (4) allow for modeling the relations between dependencies. Existing variability modeling approaches only support the first two requirements, but lack support for the latter two. The contribution of this paper is a framework for variability modeling, COVAMOF, that provides support for all four requirements.

1 Introduction

Software product families are recognized as a successful approach to reuse in software development [6] [9] [17] [22]. The philosophy behind software product families is to economically exploit the commonalities between software products, but at the same time preserve the ability to vary the functionality between these products. Managing these differences between products, referred to as variability, is a key success factor in product families [7]. Research in the context of software product families is shifting from focusing on exploiting the commonalities towards managing the variability, referred to as variability management, e.g. [1] [2] [4] [8] [13].

A key aspect of variability management is the explicit representation of the variability. Recent research [3] [5] [7] [8] [11] [21] agrees that this variability should be modeled uniformly over the lifecycle, and in terms of dependencies and first class represented variation points.

Industrial case studies we performed at organizations that employ medium and large scale software product families have shown that, in addition to providing the required functionality, the main focus during product derivation is on satisfying complex dependencies, i.e. dependencies that affect the binding of a large number of variation points, such as quality attributes. A key aspect in resolving these dependencies is having an overview on these complex dependencies and how they mutually relate. An example of a complex dependency is a restriction on memory

usage of a software system. An example of a relation to other dependencies is how this restriction *interacts* with a requirement on the performance. These case studies therefore indicate the need for the first-class representation of dependencies, including complex dependencies, in variability models and the need for means to model the relations between these dependencies.

In the past few years, several approaches have been developed for modeling the variability in software product families [3] [5] [8] [12] [18] [21]. Most of these approaches treat variation points as first class citizens and provide means to model simple (1-to-1) dependencies, and some of the approaches model the variability uniformly over the lifecycle. None of the approaches, however, supports the modeling of complex dependencies.

In this paper, we present COVAMOF, a variability modeling approach that uniformly models the variability in all abstraction layers of the software product family. It treats variation points and dependencies as first-class citizens and provides means to model the relations between simple and complex dependencies. We illustrate this approach with an industrial case study.

The structure of this paper is as follows: in the next section we present an industrial case study that serve as a guiding example throughout this paper. In section 3 we present the problem statement. Section 4 presents our approach to variability modeling, COVAMOF, whereas section 5 discusses this approach and concludes the paper.

2 Case Description

In order to get a better insight in families, we performed three industrial case studies at organizations that employ medium and large scale software product families in the context of software intensive technical systems, i.e. Robert Bosch GmbH, Thales Naval Netherlands B.V., and Dacolian B.V. All three cases served as basis for the contents of this paper. In [11], we describe the first two case studies in detail. In this section, we introduce the case study we performed at Dacolian B.V. This case is used to exemplify the issues and our COVAMOF framework.

2.2 Dacolian B.V.

Dacolian B.V. is an independent SME in the Netherlands that mainly develops intellectual property (IP)-software modules for intelligent traffic systems (ITS). Dacolian does not sell its products directly to end users. Their customers are internationally operating companies that are market leaders within a part of ITS market. Products of Dacolian's Intrada® product family are often essential parts in the end product of these companies.

In our case study we focused on the Intrada® product family whose members use outdoor video or still images as most important sensor input. Intrada products deliver, based on these real-time input images, abstract information on the actual contents of the images. Examples are: detection of moving traffic, vehicle type classification, license plate reading, video based tolling and parking.

Dacolian maintains a product-family architecture and in-house developed reusable components that are used for product construction. Many of these components are frameworks. Examples are a framework providing a common interface to a number of different frame grabbers (real and virtual) (See Figure 1: Image intake), and a framework providing license specialization and enforcement (See Figure 1: License Manager). The infrastructure also contains special designed tooling for Model Driven Architecture (MDA) based code generation, software for module calibration, dedicated scripts and tooling for product, feature and component testing, and large data repositories. These assets capture functionality common to either all Intrada products or a subset of Intrada products.

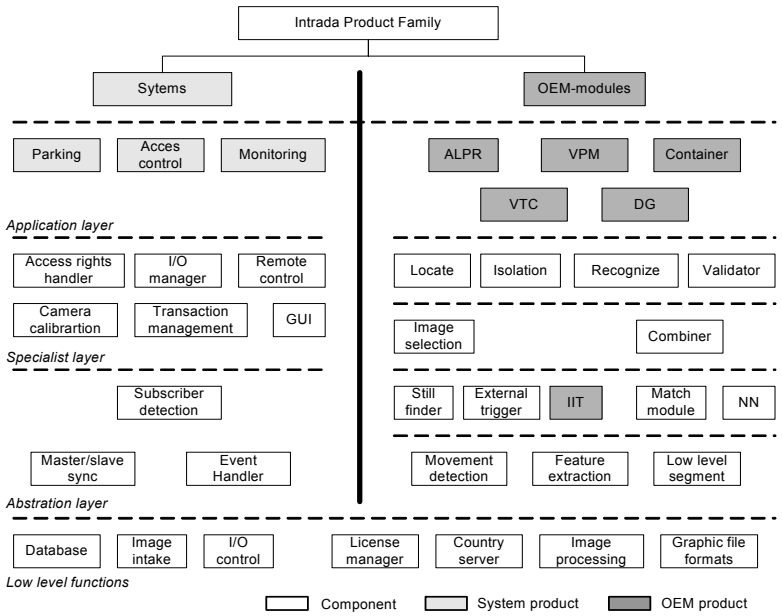

Fig. 1. .The Logical view on the Intrada Product Family architecture

Figure 1 presents the logical view of the architecture of the Intrada product family. This logical view contains a number of components. The responsibilities of these components are denoted by the layers in the logical view. Each component incorporates variability and uses some of the components in a lower abstraction level. The thick solid vertical line gives the separation between the two branches within the Intrada product family. The low level functions layer provides basic functionality, hides hardware detail and gives platform independence. The abstraction layer adds semantic information to the raw image data. The specialist layer contains core components of Dacolian's Intrada product family. The blocks in this layer perform the complex calculations. The top layer defines the various products. The grey shaded blocks indicate products that are sold by Dacolian B.V. The OEM-modules (dark grey) have no interaction with hardware components, whereas the system products (light grey) like parking, access control, and monitoring include components that communicate directly with the real world.

For products at the upper layer, binding is typically by configuration and license files at startup time for the products at the upper layer. Lower layers use binding mechanisms that bind before deployment. Typical examples are MDA code generation, precompiler directives and Make dependencies. Binding at link time is almost not practiced in the Intrada product family, and Make dependencies only specify variation at compile time.

3 Problem Statement

In section 1, we indicated the explicit representation of variability of software product families as a key aspect of variability management. In this section, we discuss variability in terms of variation points and variability dependencies, and present requirements on variability modeling techniques. We furthermore verify to which extent existing approaches address these requirements.

3.1 Variation Points

Variation points are places in a design or implementation that identify locations at which variation occurs [15]. These identifiers have been recognized as elements that facilitate systematic documentation and traceability of variability, development for and with reuse [8], assessment and evolution. As such, variation points are not by-products of design and implementation of variability, but are identified as central elements in managing variability. Figure 2 shows that the hierarchy in product families is divided into three abstraction layers, i.e. features, architecture, and component implementations, and that the hierarchy throughout these layers is defined by levels of abstraction. As variation in software product families can occur in all three abstraction layers, variation points can also be identified in all those layers. Variation points in one abstraction layer can *realize* the variability in a higher abstraction level, e.g. an optional architectural component that realizes the choice between two features in the feature tree.

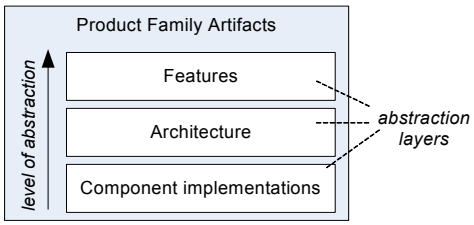

Fig. 2. Hierarchy in product families. The product family is divided into three abstraction layers, i.e. features, architecture, and component implementations and the hierarchy throughout these layers is defined by levels of abstraction

Each variation point is associated with zero or more variants and can be categorized to five basic types.

- An *optional* variation point is the choice of selecting zero or one from the one or more associated variants.
- An *alternative* variation point is the choice between one of the one or more associated variants.
- An *optional variant* variation point is the selection (zero or more) from the one or more associated variants.
- A *variant* variation point is the selection (one or more) from the one or more associated variants.
- A *value* variation point is a value the can be chosen in a predefined range.

A variation point can be in two different *states*, i.e. open or closed. An open variation point is a variation point to which new variants can be added. A closed variation point is a variation point to which it is not possible to add new variants. The state of a variation points may change from one development phase to the next.

All variability in software product families is realized by variation points in a lower layer of abstraction or implemented by a *realization technique* in the product family artifacts. Over the past few years, several of these variability realization techniques have been identified for all three abstraction layers. Examples of these mechanisms include architectural design patterns, aggregation, inheritance, parameterization, overloading, macros, conditional compilation, and dynamically linked libraries, see also, e.g. [15] and [1]. [20] presents a characterization of realization techniques. Realization techniques in the artifacts impose a *binding time* on the associated variation point, i.e. the development phase at which the variation point should be bound, e.g. the binding time of a variation point that is implemented by condition compilation is the compilation phase.

3.2 Variability dependencies

Dependencies in the context of variability are restrictions on the variant selection of one or more variation points, and are indicated as a primary concern in software product families [16]. These dependencies originate, amongst others, from the application domain (e.g. customer requirements), target platform, implementation details, or restrictions on quality attributes.

In product families, there are several types of dependencies. Simple dependencies specify the restriction on the binding of one or two variation points. These dependencies are often specified in terms of *requires* and *excludes* relations and are expressed along the lines of “the binding of variant A1 to variation point A excludes the binding of variant B1 to variation point B”. Experience in industrial product families showed that, in general, dependencies are more complex and typically affect a large number of variation points. Dependencies can, in many cases, not be stated formally, but have a more informal character, e.g. “these combinations of parameter settings will have a negative effect on the performance of the overall software system”.

In some cases the validity of dependencies can be calculated from the selection of variants of the associated variation points and the influence of a reselection of an associated variation point on the validity can be predicted. We refer to these dependencies as *statically analyzable* dependencies. Examples of these dependencies are the mutual exclusion of two component implementations or a limitation on the memory usage of the whole system (in the situation the total memory usage can be computed from the variant selections).

In other cases, dependencies cannot be written down in such a formal way. The validity of the dependency therefore cannot be calculated from the variant selections and the verification of the validity of the dependency may require a test of the whole software system. We therefore refer to these dependencies as *dynamically analyzable* dependencies. In most cases, software engineers can predict whether a reselection of variants will positively or negatively affect the validity of the dependency. However, in some cases software engineers cannot predict the influence of a reselection of variants at all. We found examples of these dependencies at both Thales Naval Netherlands and Dacolian B.V., where software engineers indicate that a requirement on the overall performance can only be validated by testing the whole software product.

Dependencies are not isolated entities. The process of resolving one dependency may affect the validity of other dependencies. We refer to this as *dependency interaction*. Dependencies that mutually interact have to be considered simultaneously and resolving both dependencies often requires an iterative approach. As the verification of dependencies may require testing the whole configuration, solving dependencies that mutually interact is one of the main concerns for software engineers and hampers the product derivation process.

3.3 Requirements

Making the variability in software product families explicit has been identified as an important aspect of variability management [8] [11]. Below, we present four requirements that we recognize as important for a variability modeling technique. These requirements originate from related research, the case studies, and our experience in software product families.

R1. Uniform and first-class representation of variation points in all abstraction levels. The uniform and first-class representation of variation points facilitates the assessment of the impact of selections during product derivation and changes during evolution [7].

R2. Hierarchical organization of variability representation. [11] reports on the impact of large numbers (hundreds of thousands) of variation points and variants on the product derivation process in industrial product families. Explicitly organizing these variation points reduces the cognitive complexity that these large numbers impose on engineers that deal with those variation points.

R3. Dependencies, including complex dependencies, should be treated as first class citizens in the modeling approach. Industrial experience indicates that most effort during product derivation is on satisfying complex dependencies, e.g. quality

attributes. As the first class representation of dependencies can provides a good overview on all dependencies in the software product family, the efficiency of product derivation increases.

R4. The interactions between dependencies should be represented explicitly. For efficient product derivation, software engineers require an overview on the interactions between dependencies to decide which strategy to follow when solving the dependencies. Therefore, these interactions should be modeled explicitly.

3.4 Related work

As we briefly mentioned in the introduction, several approaches to variability modeling exist. Below, we briefly present these approaches, and verify to what extent each approach satisfies the aforementioned requirements.

A Meta-model for Representing Variability in Product Family Development
Bachman et al. [3] present a meta-model for representing the variability in product families, that consists of variation points in multiple views. These views correspond to the layers of the product family we introduced in section 3. As the meta-model consists of variation points that are refined during development, it is clear how choices map between layers of abstraction. This model does not provide a hierarchical organization of variation points and dependencies are not treated as first class citizens. It does provide means to model 1-to-n dependencies between variation points.

Mapping Variabilities onto Product Family Assets
Becker [5] presents an approach in which the representation of variability of the product family is separated into two levels, i.e. the specification and the realization level. The variability on the specification level is defined in terms of variabilities, and on the realization level in terms of variation points. Variabilities specify the required variability and variation points indicate the places in the asset base that implement the required variability. This model contains two types of these variation points, i.e. static (pre-deployment), and dynamic (post-deployment) variation points. Dependencies can be specified in a 1-to-1 manner and are not represented as first-class citizens.

Generic Modelling using UML Extensions for Variability
Clauss [8] presents an approach to model the variability of product families by extending UML models. Both the feature models and the design models can be extended with variability. Variation points in these models are marked with the stereotype <<variationPoint>> on features and components and not treated as first class citizens. Dependencies are modeled as constraints between two variants, and can therefore be associated to one or two variation points and are not represented as first-class citizens.

Multiple-View Meta-Modeling of Software Product Lines
Gomaa and Shin [12] present an extension to UML models to capture the variability of the product family on the feature and design level. Variation points are not treated

as first class citizens but rather defined implicitly by marking features or classes as optional or variant. Dependencies are modeled as first-class citizens by dependency meta-classes and restrict the selection of two variants.

A Composable Software Architecture for Consumer Electronics Products
Van Ommering [18] presents how Koala [19] can be used in the context of software product families. Koala is an approach to realize the late binding of components. Components are recursively specified in terms of first class provided and required interfaces. The variability in the design is specified in terms of the selection of components, parameters on components, and the runtime routing of function calls. Dependencies are specified by the interfaces of components, and define, amongst others, whether two components are compatible.

Systematic Integration of Variability into Product Line Architecture Design
In their paper on architectural variability, Thiel and Hein [21] propose an extension to the IEEE P1471 recommended practice for architectural description [14]. This extension includes using variation points to model variability in the architecture description. Variation points in the variability model include aspects such as binding time, one or more dependencies to other variation points, resolution rules that captures, for example, which architectural variants should be selected with which options.

3.5 Summary

Table 1 presents a summary of the existing variability approaches and their adherence to the requirements. Some of the approaches support the up to two requirements, but none of them supports all four. In the next section, we present COVAMOF, an approach to variability modeling that addresses all four requirements.

Table 1. The adherence of approaches to variability modeling to the requirements in section 3.3

	R1	R2	R3	R4
Bachman et al. [3]	+ +	+ +	-	- -
Becker [5]	+	+ +	-	- -
Clauss [8]	-	+	- -	- -
Gomaa and Shin [12]	-	+/-	+/-	- -
van Ommering [18]	-	+	- -	- -
Thiel and Hein [21]	-	+/-	-	- -

4 ConIPF Variability Modeling Framework (COVAMOF)

During the ConIPF project [10], we developed the ConIPF Variability Modeling Framework (COVAMOF). COVAMOF consists of the COVAMOF Variability View (CVV), which provides a view on the variability provided by the product family

artifacts. The COVAMOF Variability View can be derived from the product family artifacts manually or automatically and we are currently working on tool support for automatic derivation of this view. The CVV encompasses the variability of artifacts on all layers of abstraction of the product family (from features down to code), as shown in Figure 3. This figure also shows that the COVAMOF defines two views on the CVV, i.e. the Variation Point View, and the Dependency View. In the following subsection, we describe the structure of the CVV. In subsection 4.2, we show how the two views of CVV are used during product derivation.

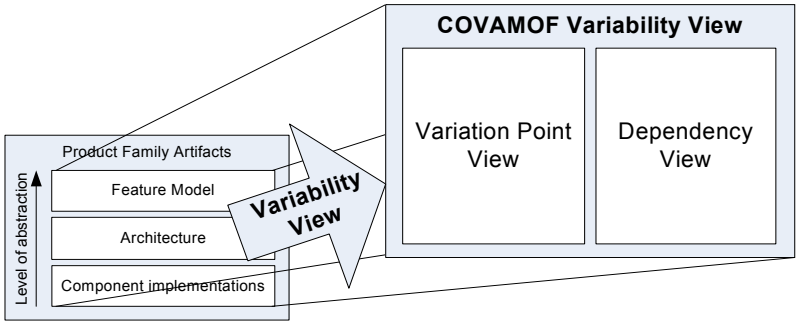

Fig. 3. COVAMOF provides the COVAMOF Variability View (CVV) on the provided variability of the product family artifacts on all layers of abstraction. In addition, COVAMOF defines two views on the CVV, i.e. the Variation Point View and the Dependency View

4.1 COVAMOF Variability View (CVV)

The CVV captures the variability in the product family in terms of variation points and dependencies. The graphical notations of the main entities in the CVV are shown in figure 4. We describe the entities, and relations between them, in more detail below.

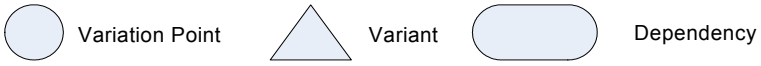

Fig. 4. Graphical notations of variation points, variants, and dependencies in the CVV

Variation Points
Variation points in the CVV are a view on the variation points in the product family. Each variation point in the CVV is associated with an artifact in the product family, e.g. a feature tree, a feature, the architecture, or a C header file. In correspondence to section 3.1, there are five types of variation points in the CVV, i.e. optional, alternative, optional variant, variant, and value. Figure 5 presents the graphical representation of these types.

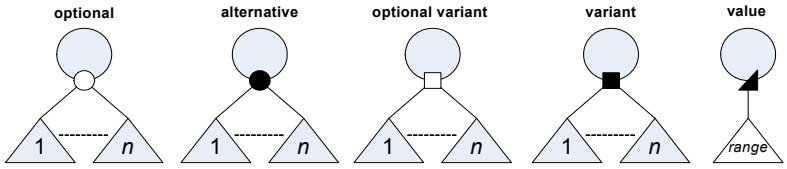

Fig. 5. The graphical notation of the five types of variation points in the CVV

The variation points in the CVV specify, for each variant or value, the actions that should be taken in order to realize the choice, for that variant or value, in the product family artifacts, e.g. the selection of a feature in the feature tree, the adaptation of a configuration file, or the specification of a compilation parameter. These actions can be specified formally, e.g. to allow for automatic component configuration by a tool, or in natural language, e.g. a guideline for manual steps that should be taken by the software engineers.

A variation point in the CVV furthermore contains a *description* and information about its *state* (open or closed), the *rationale* behind the binding, the *realization mechanism*, and its associated *binding time* (where applicable). The *rationale* behind the binding defines on what basis the software engineer should make his choice between the variants, in the case the information from associated realization relations (see below) does not suffice.

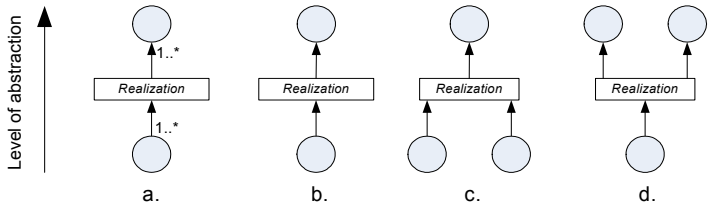

Fig. 6. The possible number of variation points associated with a realization relation (a.), a variation point realizing one other variation point on a higher level of abstraction (b.), two variation points realizing one variation point (c.), and one variation point realizing two variation points (d.)

Realization relations

Variation points that have no associated realization mechanism in the product family artifacts are realized by variation points on a lower level of abstraction. These realizations are represented by *realization relations* between variation points in the CVV (See figure 6a). The realization relation in the CVV contains rules that describe how a selection of variation points directly depends on the selection of the variation points in a higher level of abstraction in the product family. Figure 6 also presents three typical examples of realization relations between variation points. In practice, any combination of these realization relations exists in software product families. Realization relations between variation points imply that the binding time of the variation points being realized depend on their realization at a lower level of abstraction. These realization relations not only define the realization between levels

of abstraction within one abstraction layer, but also define how variation points in one abstraction layer (e.g. in the architecture) realize variation points in a higher abstraction layer (e.g. in the features).

We present three examples from the case studies that illustrate the usage of variation points and realization in the CVV.

Example: Figure 7 presents an example of one variation point realizing one other variation point. A feature of the Intrada Product family is the image source. The offered alternatives are digital/analog video input, files stored on a harddisk, or pointers to chuncks of memory that contain the image data. VP1 shows the available variants of which one should be selected as image source. This variation point is realized by variation point VP2, which offers two variants that accommodate for the alternatives of VP1.

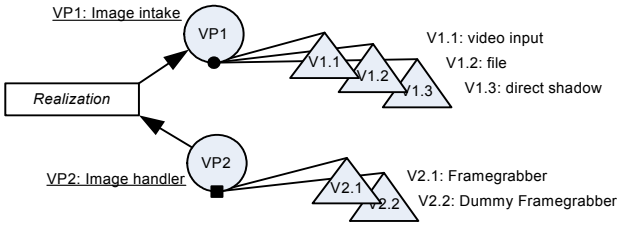

Fig. 7. Realization example – one variation point realizing one other variation point at a higher level of abstraction

Example: Figure 8 presents an example of multiple variation points realizing one variation point. Many Intrada products can be configured to recognize license plates from one or more countries simultaneously. This feature is expressed by the variant variation point VP3, where each variant represents one country. At a lower level, VP3 is realized by three variation points, VP4, VP5, and VP6. Which variants of VP4, VP5, and VP6 are included in the derived products depends on the logical "requires"-dependencies between VP3 and the before mentioned variation points. VP5 is an optional variant variation point and its variants are only used when there is no speed limitation to improve the performance.

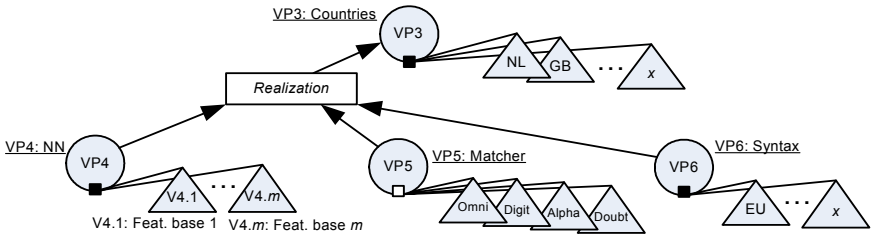

Fig. 8. Example of multiple variation points realizing one variation point

Example: Figure 9 presents an example of one variation point realizing two variation points. The parking, access control, and monitoring products allow access rights differentiation. For the three Intrada products this is respectively subscriber specific,

group specific, or no differentiation. Within the product family this is represented by a variation point (VP7). Also various methods of transaction logging, statistic, and error reporting can be selected (VP8). Both VP7 and VP8 are realized by VP9 which provides database and search facilities.

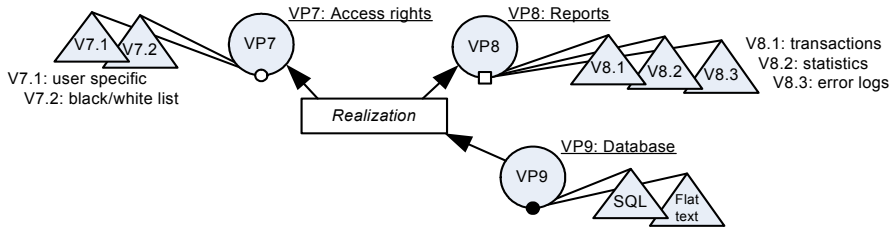

Fig. 9. Example of one variation point realizing two variation points at a higher level

Dependencies

Dependencies in the CVV are associated with one or more variation points in the CVV and restrict the selection of the variants associated to these variation points. The properties of dependencies consist of the *type* of dependency, a *description* of the dependency, the *validation time*, and *the types of associations* to variation points. The validation time denotes the development stage at which the validity of the dependency can be determined with respect to the variation point binding. We describe the type of dependency and types of association in detail below.

Types of associations

On the level of dependencies, the CVV distinguishes between three types of associated variation points. The type of the association of a variation point depends on the available knowledge about the influence of the variant selection on the validity of the dependency. (See also figure 10).

Predictable associations represent variation points whose influence of the variant selection on the validity of the dependency is fully known. The impact of variant selection on the validity of the dependency can be determined before the actual binding of the selected variant(s).

Directional associations represent variation points whose influence of the variant selection on the validity of the dependency is not fully known. Instead, the dependency only specifies whether a (re)selection of variants will either positively or negatively affect the validity of the dependency.

Unknown associations represent variation points of which it is known that the variant selection influences the validity of the dependency. However, the dependency does not specify *how* a (re)selection of the variants influences the validity.

In section 3.2, we introduced the distinction between statically analyzable and dynamically analyzable dependencies. Note that statically analyzable dependencies correspond to dependencies in the CVV that only have predictable associations. Dynamically analyzable dependencies correspond to dependencies in the CVV that have at least one directional or unknown association.

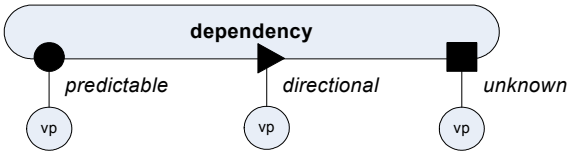

Fig. 10. The three types knowledge about the influence of the associated variation points on the dependency

Types of dependencies

The CVV distinguishes between three types of dependencies, i.e. logical, numerical and nominal dependencies. Figure 11 shows the graphical notation of each type of dependency.

Logical dependencies specify a function *valid*, which yields the validity of the dependency for each selection of variants of the associated variation points. This can for example be the mutual exclusion of two component implementations. As the influence of each associated variation point is defined by the *valid* function, variation points are only predictably associated to logical dependencies. Therefore, logical dependencies can always be analyzed statically.

Numerical dependencies define a numerical value *N*. The value of N depends on the variants selection of the associated variation points. The validity of the dependency is expressed by the specification of a valid range on the numerical value N. An example of a numerical dependency is the restriction of the total memory consumption of a software system to 64 megabytes. These dependencies can have all three types of associated variation points. The new value of N after a (re)selection of a predictable associated variation point can be calculated. The dependency specifies whether the (re)selection of a directional associated variant will increase or decrease the value of N. It does not specify the influence of unknown dependencies.

Nominal dependencies specify a set of categories. The binding of all variation points associated to the dependency map to one of the categories. The validity of the dependency is expressed by the specification of the valid categories. There are only unknown associated variation points associated to nominal dependencies.

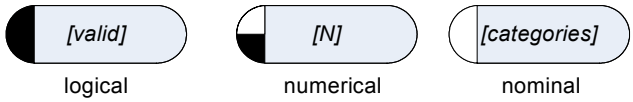

Fig. 11. The graphical notation of the three types of dependencies and three types of associations to variation points

In order to exemplify the usage of dependencies in the CVV, we provide two examples from the case studies.

Example: Figure 12a presents the representation of a logical dependency “countries”. This dependency captures the complex logical relations that exist between the variants of the involved variation points VP3, VP4, VP5, and VP6. Rules that are represented by this dependency include amongst others: “variant V3.NL

requires variant V4.1 and V6.EU, variant V4.3 excludes variant V4.4." (See figure 8 for more details on the involved variation points.)

Example: Figure 12b presents the representation of a numerical dependency "error rate". The possible values for the "error rate" that can be obtained by the Intrada product family depend on the selected feature bases (VP4), the included matchers (VP5), and the settings of the various range parameters associated with the Isolate variation point (VP10). Only for the variants of the matcher variation point is known whether it will improve or worsen the "error rate" score. The actual influence of the two other variation points is unknown and can only be determined by test procedures.

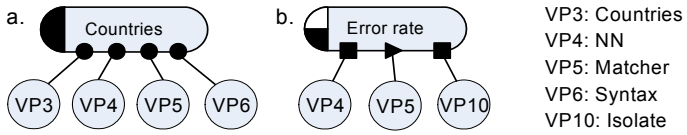

Fig. 12. An example from the case study of the representation of a logical dependency (a.) and a numerical dependency (b.)

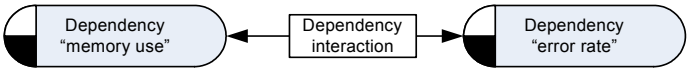

Fig. 13. An example of a dependency interaction from the case study

Dependency interactions

Dependency interactions in the CVV specify how two or more dependencies mutually interact. The interaction provides a description of the origin of the interaction and specifies how to cope with the interaction during product derivation.

Example: In order to illustrate the dependency interactions in the CVV, we present an example from the case study in figure 13. There is a mutual interaction between the two numerical dependencies "memory use" and "error rate". Both dependencies include the variation point "matcher" (see figure 8). Including additional matcher variants will improve the "error rate" dependency but downgrade the "memory use" dependency. Individual scores on each dependency and the costs of the associated quality attributes will determine the order in which the dependencies should be handled.

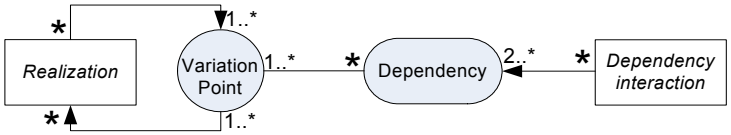

Fig. 14. The main entities in the CVV Meta-model

Notation

The main entities in the CVV meta-model are presented in figure 14. It summarizes the CVV: variation points in the CVV realize other variation points in a n-to-m manner and are associated to zero or more dependencies. Dependencies are associated to one or more variation points, and dependency interactions define relations between two or more dependencies.

We use a XML based language, CVVL (CVV Language), to represent the CVV in text. Variation points and dependencies are represented as first-class entities, and its notation is illustrated in figure 15 and 16.

```
<variationpoint id="[id]">
  <artefact>[artefact identifier]</artefact>
  <abstractionlayer>[abstraction layer]</abstractionlayer>
  <description>[description]</description>
  <type>optional|alternative|optional variant|variant|value<type>
  <variants> <!-- if not type=value -->
    <variant id="[id]">
    . . .
    <variant id="[id]">
  </variants>
  <range>[range specification]</range> <!-- if type=value -->
  <state>open|closed</state>
  <mechanism>[mechanism]</mechanism>
  <bindingtime>[bindingtime]</bindingtime>
  <rationale>[rationale]</rationale>
</variationpoint>
```

Fig. 15. Notation of variation points in CVVL

```
<dependency id="[id]">
  <description>[description]</description>
  <type>logical|numerical|nominal</type>
  <validationtime>[validation time]</validationtime>
  <associations>
    <association type="predictable|directional|unknown">
      <variationpoint id="[id]">
    </association>
    . . .
    <association type="predictable|directional|unknown">
      <variationpoint id="[id]">
    </association>
  </associations>
  <logical>  <!-- if type=logical -->
   <valid>[valid function]</valid>
  </logical>
  <numerical> <!-- if type=numerical -->
    <N>[description and origin of N]</N>
    <range>[valid range of N]</range>
  </numerical>
  <nominal>  <!-- if type=nominal -->
    <categories>
       <category valid="yes|no">[category]</category>
         . . .
       <category valid="yes|no">[category]</category>
    </categories>
  </nominal>
</dependency>
```

Fig. 16. The notation of dependencies in CVVL

4.2 Development Views

In addition to the CVV described above, COVAMOF also defines two development views on the CVV, i.e. the Variation Point View and the Dependency View (See also

figure 17). These views enable software engineers to shift the focus between variation points and dependencies, in order to develop a strategy to derive products.

Variation Point View
The Variation Point View in the CVV provides the software engineers with an overview on the variation in all abstraction levels of a product family in terms of variation points. The realization relations provide the structure on the set of variation points and the dependencies in this view are attributes of the variation points.

Dependency View
The main entities in the Dependency View are the dependencies and the dependency interactions that provide the structure on the set of dependencies. Variation points in this view are attributes of the dependencies. The Dependency View provides software engineers with an overview on the most critical dependencies, e.g. based on their type, number of associated variation points or number of dependency interactions, which can be used to develop a strategy to resolve the dependencies during product derivation.

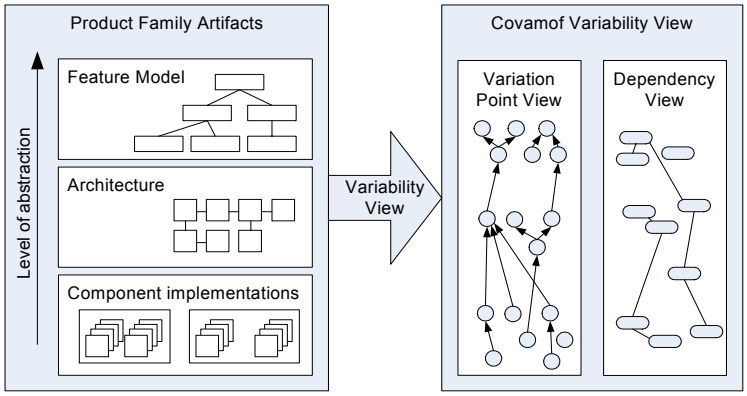

Fig. 17. COVAMOF provides two development views on the artifacts, i.e. the Variation Point View and the Dependency View

5 Conclusion and future work

In this paper, we presented four requirements that we deem important for modeling variability in the context of software product families. We showed that none of the related approaches addresses all requirements. In response to the observed limitations, we have developed COVAMOF, a framework for modeling variability in software product families. Below, we show how COVAMOF addresses these requirements.

R1. Uniform and first-class representation of variation points in all abstraction levels. COVAMOV provides a view, the CVV, in which the variability of features, architecture and component implementations are all modeled, and uniformly represented by variation points as first class citizens. COVAMOF therefore fully satisfies requirement R1.

R2. Hierarchical organization of variability representation. In the CVV, the realization relation defines how variation points realize variation points at higher level of abstraction. As these realization relations provide a hierarchical organization of the variability, COVAMOF fully satisfies requirement R2.

R3. Dependencies, including complex dependencies should be treated as first class citizens in the modeling approach. COVAMOV provides a view, the CVV, in which dependencies are modeled as first class citizens. These dependencies restrict the selection of one or more variation points. COVAMOF therefore fully satisfies requirement R3.

R4. The interactions between dependencies should be represented explicitly. The dependency relation defines which dependencies should be considered simultaneously and whether one dependency should be solved before another dependency. COVAMOF therefore fully satisfies requirement R4.

A well-known problem with modeling is the issue of how to keep the model consistent with the artifacts. As already mentioned in section 4, the COVAMOF Variability View can be derived from the product family artifacts manually or automatically. We are currently working on the *intrinsic modeling* of the variability in the artifacts and the required tooling support for the automatic derivation of the model. *Intrinsic modeling* means that the product family artifacts themselves capture their own variability model (in terms of variation points and dependencies) and that each variability realization technique in the artifacts provides the variability modeling elements. With respect to COVAMOF, these variability modeling elements in intrinsically modeled product families *are* the CVV and changes to the product family artifacts *are* changes to the CVV. Note that this addresses the problem of keeping the model consistent with the artifacts.

Acknowledgements. This research has been sponsored by ConIPF [10] (Configuration in Industrial Product Families), under contract no. IST-2001-34438. The ConIPF project aims to define and validate methodologies for product derivation that are practicable in industrial applications. We thank the industrial partners, in particular Dacolian B.V., for their valuable input.

6 References

1. M. Anastasopoulos, C. Gacek: Implementing product line variabilities. In: Symposium on Software Reusability (SSR'01), Toronto, Canada, Software Engineering Notes 26 (3) 109–117, 2001.
2. F. Bachmann, L. Bass: Managing variability in software architecture. In: Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'01), pp. 126–132, 2001.

3. F. Bachman, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, A. Vilbig: Managing Variability in Product Family Development, accepted for the 5th Workshop on Product Family Engineering (PFE-5), November 2003, to be published in Springer Verlag Lecture Notes on Computer Science, 2004.
4. D. Batory, S. O'Malley: The Design and Implementation of Hierarchical Software Systems with Reusable Components, ACM Transactions on Software Engineering and Methodology, 1(4): pp. 355-398, October 1992.
5. M. Becker: Mapping Variability's onto Product Family Assets, Proceedings of the International Colloquium of the Sonderforschungsbereich 501, University of Kaiserslautern, Germany, March 2003.
6. J. Bosch: Design & Use of So ftware Architectures, Adopting and Evolving a product-line approach, Addison-Wesley, ISBN 0-201-67494-7, 2000.
7. J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, K. Pohl: Variability Issues in Software Product Lines, Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), pp. 11–19, 2001.
8. M. Clauss: Modeling variability with UML, GCSE 2001 - Young Researchers Workshop, September 2001.
9. P. Clements, L. Northrop: Software Product Lines: Practices and Patterns, SEI Series in Software Engineering, Addison-Wesley, ISBN: 0-201-70332-7, 2001.
10. The ConIPF project (Configuration of Industrial Product Families), http://www.rug.nl/conipf.
11. S. Deelstra, M. Sinnema, J. Bosch: Product Derivation in Software Product Families; A Case Study, accepted for the Journal of Systems and Software, 2003.
12. H. Gomaa, M.E. Shin: Multiple-View Meta-Modeling of Software Product Lines, 8th International Conference on Engineering of Complex Computer Systems (ICECCS 2002), IEEE Computer Society 2002, ISBN 0-7695-1757-9, pp. 238-246, 2002.
13. L. Griss, J. Favaro, M. d'Alessandro: Integrating feature modeling with the RSEB, Proceedings of the Fifth International Conference on Software Reuse (Cat. No. 98TB100203), IEEE Computing Society, xiii+388, pp.76-85, 1998.
14. IEEE Recommended Proactice for Architectural Description of Software-Intensive Systems (IEEE Standard P1471), IEEE Architecture Working Group (AWG), 2000.
15. I. Jacobson, M. Griss, P. Jonsson: Software Reuse. Architecture, Process and Organization for Business Success. Addison-Wesley, ISBN: 0-201-92476-5, 1997.
16. M. Jaring and J. Bosch: Variability Dependencies in Product Family Engineering, accepted for the 5th Workshop on Product Family Engineering (PFE-5), November 2003, to be published in Springer Verlag Lecture Notes on Computer Science, 2004.
17. M. Jazayeri, A. Ran, F. van der Linden: Software Architecture for Product Families: Principles and Practice, Addison-Wesley, 2000.
18. R. van Ommering: A Composable Software Architecture for Consumer Electronics Products, XOOTIC Magazine, March 2000, Volume 7 number 3, 2000.
19. R. van Ommering, F. van der Linden, J. Kramer, J. Magee: The Koala Component Model for Consumer Electronics Software, IEEE Computer, p78-85, March 2000.
20. M. Svahnberg, J. Gurp, J. Bosch: A Taxonomy of Variability Realization Techniques, technical paper ISSN: 1103-1581, Blekinge Institute of Technology, Sweden, 2002.
21. S. Thiel, A. Hein: Systematic integration of Variability into Product Line Architecture Design, Proceedings of the 2nd International Conference on Software Product Lines (SPLC-2), August 2002.
22. D.M. Weiss and C.T.R. Lai: Software Product-Line Engineering: A Family Based Software Development Process, Addison-Wesley, ISBN 0-201-694387, 1999.