

# Managing Variability in Software Product Families

M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch

*University of Groningen, Department of Mathematics and Computing Science,*

*P.O. Box 800, 9700 AV The Netherlands*

*{m.sinnema | s.deelstra | j.nijhuis | j.bosch}@cs.rug.nl*

## Abstract

*Software product families have proven to be an effective approach to reuse in software development. In contrary to popular belief, however, deriving products from a product family in an industrial context is an expensive and time-consuming activity. Modelling the variability in a product family can address most of the underlying problems. COVAMOF is a variability modelling approach that represents variation points and dependencies on all abstraction layers as first class citizens, supports the modelling of relations between dependencies, provides traceability and a hierarchical organization of variability throughout all abstraction layers, and supports intrinsic variability modelling. In this paper, we present COVAMOF, its tool support and its formal notation. We furthermore show the benefits of our approach over the existing approaches to variability modelling.*

## 1. Introduction

Software product families [4] [7] are recognized as an effective approach to reuse in software development. The basic reuse philosophy of software product families is intra-organizational reuse through the explicitly planned exploitation of the commonalities between related products. This philosophy has been adopted by a wide variety of organizations and has proven to substantially decrease costs and time-to-market, and increase the quality of their software products. Although this approach has been applied successfully at several organizations, several challenges remain.

In [9] we presented problems and issues we identified in large scale product families during industrial case studies and showed that deriving individual products from shared software artefacts is, in contrary to popular belief, a time-consuming and

expensive activity. The main underlying problem is the complexity of variability in product families and that organizations lack of methodologies to manage this complexity. Modelling the variability is a key aspect in addressing these issues [5] [6]. In [5] and [9] we show that approaches to variability modelling require first-class representation of variation points and dependencies, on all abstraction levels, and in all phases of the development life-cycle. Those approaches should furthermore address the problem of the possible drift between variability models and the product family artefacts [14].

In the past few years, several approaches have been developed for modelling the variability in software product families [1] [2] [3] [6] [13] [16] [20] [25]. In this paper, we show that none of these approaches address all problems. The ConIPF Variability Modelling Framework (COVAMOF) is our approach to variability that models variability on all levels of abstraction in terms of first class variation points and provides traceability and a hierarchical organization on the variability. COVAMOF furthermore models statically as well as dynamically analyzable dependencies and the relations between these dependencies, and addresses the problem of the possible drift between the model and the artefacts. COVAMOF therefore does address all requirements.

The contribution of this paper is a description COVAMOF, its tool support, and its formal notation. The structure of this paper is as follows: in the next section we present the background and related work of variability modelling. The problems we have identified in industrial product families are presented in section 3, and in section 4 we briefly present the product families we used in the case studies. In section 5, we present COVAMOF and its tool support, and in 6 we show how we validated this framework. We conclude this paper in section 7.

## 2. Background and related work

With the term variability, we refer to the ability of a software system or artefact to be configured, customized, extended, or changed for use in a specific context. Variability can occur on all layers of abstraction, e.g. in features, the architecture, and the component implementations. Examples of variability are the choice between two features, the choice between multiple architectural styles and a parameter setting of an executable. Managing this variability has been identified as a key success factor in the exploitation of software product families [5].

Variation points are places in a design or implementation that identify locations at which a choice can be made between zero or more variants. These identifiers have been recognized as elements that facilitate systematic documentation and traceability of variability, development for and with reuse [6], 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.

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 [19]. These dependencies originate, amongst others, from the application domain (e.g. customer requirements), target platform, implementation details of restrictions on quality attributes.

Making the variability in software product families explicit is an important aspect of variability management [6] [9]. In the past few years, several approaches to variability modelling have been developed in the context of software product families. Below, we briefly introduce the existing approaches.

**A1.** Bachman et al. [2] present a meta-model for representing the variability in product families, which 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.

**A2.** Becker [3] 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.

**A3.** Clauss [6] 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 modelled as constraints between two variants, and can therefore be associated to one or two variation points and are not represented as first-class citizens.

**A4.** Gomaa and Shin [13] 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 modelled as first-class citizens by dependency meta-classes and restrict the selection of two variants.

**A5.** Van Ommering [20] presents how Koala [21] can be used in the context of software product families. Koala is an approach to realize the uniform binding of pre-runtime and runtime variability of components, independent to its context. 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. The interfaces of Koala components specify whether two components are compatible. There is currently no support for multiple views on the product family variability.

**A6.** In their paper on architectural variability, Thiel and Hein [25] propose an extension to the IEEE P1471 recommended practice for architectural description [17]. 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.

**A7.** Hotz and Krebs [16] specify the knowledge about the variability of a product family in a structure-oriented logic-based representation language. Tool support mainly focuses on deriving product specific models from the product family knowledge base, including inference of choices and constraints and model validation.

**A8.** Asikainen et al. [1] have developed Koalish, an extension to Koala [21] with constructs to specify variability and constraints, and the possibility to select the type and number of components. The constraints can be used to specify logical dependencies between components and interfaces. Tools currently support Koalish for the application engineering stage to configure a Koalish model and generate a configuration to a Koala product model. There is currently no support for multiple views on the product family variability.

### 3. Problem statement

From our experience with several organizations that employ software product families, we have learned that deriving individual products from shared software artefacts is a time-consuming and expensive activity. In earlier work [9], we identified a number of causes of this issue. In this section, we present a number of problems that are related to these causes. These problems originate from related research, the case studies we presented in [9], and our experience with other software product families. Based on these problems, we present requirements for modelling variability, and show to what extent the variability modelling approaches presented in the previous section support these requirements.

#### 3.1 Problems

Below we present the main problems in industrial product family engineering, numbered P1 to P7.

**P1.** For both the application engineering process and the domain engineering process, an overview on the variability provided by the product family is crucial. Software engineers, however, indicate that they often lack this overview, and most information required is not available.

**P2.** In [9] we report on the impact of large numbers, i.e. tens of thousands, of variation points and variants on the product derivation process in industrial product families. Even in the situation that all variability is explicitly modelled, the complexity behind the choices remains a source of errors during

product derivation and therefore hampers application engineering. On the other hand, this complexity makes it hard to assess the variability provided by the product family with respect to new requirements.

**P3.** Besides managing the variability information of a system, the actual selection and binding of variants in a large and complex system can be a tedious and error-prone task. All tasks that can be automated may therefore significantly reduce costs and increase product derivation efficiency. However, industrial product family engineering lacks tools for product derivation.

**P4.** During product derivation, software engineers make decisions on various levels of abstraction. In most cases, the selection of variants on lower levels, e.g. in the component implementations, depend on the selection of variants on higher levels, e.g. in the features. In most cases, this information is not explicitly available to the software engineers.

**P5.** In most cases information about simple, e.g. requires and excludes, dependencies between a variation points is available to software engineers. They lack of information about more complex dependencies and dependencies that cannot be stated formally. However, we have identified in the case studies that resolving the dynamically analyzable dependencies requires multiple iterations of testing the whole system. Dependencies are therefore a major concern during product derivation in an industrial context.

**P6.** The process of resolving one dependency may affect the validity of other dependencies, i.e. *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.

**P7.** Variability models may drift from the actual variability in the product family artefacts [14]., due to the evolution and the lack of maintenance of these models.

#### 3.2 Requirements

In order to successfully support the software product family engineering process, an approach to variability modelling should address all of the problems we discussed above. From these problems,

we therefore derive eight requirements on a variability modelling approach, numbered R1 to R8.

**R1.** The variability should be modelled in terms of first-class represented points uniformly in all abstraction levels. This facilitates the assessment of the impact of selections during product derivation and changes during evolution [5] and therefore addresses problem P1.

**R2.** The variability should be organized hierarchically. This reduces the cognitive complexity that these large numbers impose on engineers that deal with those variation points and therefore addresses problem P2.

**R3.** Software engineers should be able to manage the variability in the product family from different views. Although there is not yet a general agreement about which views are useful, the reason behind multiple views is always the same [15]: separating aspects into separate views helps people to manage complexity and therefore addresses problem P2.

**R4.** As indicated by problem P3, modelling approaches should be supported by tools in order to be used effectively. This support should encompass both the application engineering and the domain engineering phase.

**R5.** Corresponding to problem P4, the variability model should provide support for traceability, i.e. modelling of how choices on a higher level of abstraction map onto choices on a lower level of abstraction.

**R6.** Simple and complex dependencies should be represented as first-class citizens in order to provide a good overview on the dependencies in the software product family. This will increase the efficiency of product derivation and addresses problem P5.

**R7.** 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 modelled explicitly. This requirement addresses problem P6.

**R8.** In order to address problem P7, changes to the product family artefacts should enforce the corresponding changes to the model. Therefore there should be a tight coupling between the model and the artefacts.

### 3.3 Evaluation

In section 2, we presented existing approaches to variability modelling. We evaluated these approaches to determine to what extent they address the requirements presented in section 3.2. In Table 1, we show the result of this evaluation. A “+” denotes full support, while a “-“ denotes no support for the requirement. We conclude from this table that none of the existing approaches addresses all eight requirements on variability modelling in industrial product families.

**Table 1 Evaluation of existing variability modelling approaches.**

	A1	A2	A3	A4	A5	A6	A7	A8
<b>R1</b>	+	+/-	-	-	-	+	-	-
<b>R2</b>	+	+	+	+/-	+	+/-	+	+
<b>R3</b>	-	-	-	+	-	+	-	-
<b>R4</b>	-	+	+	+	+	+	+	+
<b>R5</b>	+	+	-	+	+	+	+	+
<b>R6</b>	-	-	-	+/-	-	-	-	-
<b>R7</b>	-	-	-	-	-	-	-	-
<b>R8</b>	-	-	-	-	+	-	-	+

### 4. Case studies

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 [9], we describe the first two case studies in detail and show which of the problems we presented in section 3.1 were identified at the respective organizations. The Dacolian case study serves as a basis to provide examples in this paper and to illustrate the validation presented in section 6.

Dacolian B.V. is an independent SME in the Netherlands that mainly develops intellectual property (IP)-software modules for intelligent traffic systems (ITS). 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, e.g. detection of moving traffic, vehicle type classification, license plate reading, video based tolling and parking.

## 5. COVAMOF

In section 3.3, we showed that none of the existing approaches to variability modelling supports all eight requirements presented in section 3.2. In this section, we present our variability-modelling framework, COVAMOF (ConIPF [8] Variability Modelling Framework). We illustrate the framework with examples from the product family of the Dacolian case study.

A key aspect of COVAMOF is the COVAMOF Variability View (CVV), which is a view on the variability provided by the product family artefacts. Based on the CVV, COVAMOF supports the product family engineering process on the level of domain engineering, i.e. development and evolution of the product family assets, and on the level of application engineering, i.e. derivation of products from the product family assets.

The COVAMOF Variability View (CVV) is a view on the variability provided by the product family artefacts on all layers of abstraction. Examples of these artefacts are features, architectural components and software libraries. The variability view unifies the variability of artefacts on all layers of abstraction of the product family (from features down to code) and realizes the traceability between the artefacts in the layers.

In order to comply with the requirements R1 and R6, variation points and dependencies are modelled as first class citizens in the CVV. Below, we show how these variation points and dependencies and relation between these elements are modelled in the CVV.

### 5.1 Representation of the CVV

In this section, we present the formal representation of the elements in the CVV. COVAMOF also provides a graphical representation, used by software engineers, and an XML based representation, used for communication between tools. The formal representation is based on set logic and is used for COVAMOF tool development and for statistical analysis of CVV instances. We use the following definitions for the formal representation:

- $P(A)$  is the *power set* of set  $A$ , i.e. the set of all subsets of  $A$ .
- Let  $F$  be a function and  $(\forall x \in \text{domain}(F) : F(x)$  is a set), then  $\prod(F)$  is the *generalized product* of  $F$ , i.e. the set  $\{ f \mid f \text{ is a function } \wedge \text{domain}(f) = \text{domain}(F) \wedge \forall x \in \text{domain}(f) : f(x) \in F(x) \}$ .

- Let  $f$  be a function and  $B$  a set, then  $f \upharpoonright B$  is the set  $\{ (x; y) \in f \mid x \in B \}$ .
- Let  $\text{Char}$  be a set of characters, then the set  $\text{String}$  is the set  $\{ s \in \text{Char}^n \mid 0 \leq n \}$  of all possible lists of characters.

### 5.2 Layers of abstraction

In the CVV, the variability in all abstraction layers in product families are uniformly modelled in terms of variation points and dependencies. We define the extensible set layers as:

```
LAYERS = { FEATURES, ARCHITECTURE ,  
          COMPONENT_IMPLEMENTATIONS }
```

The *features* abstraction layer is the set of all features and their compositional structure. A feature is a logical unit of behaviour that is specified by a set of functional and quality requirements and is used to group and abstract from requirements [4].

The *architecture* abstraction layer contains the variability provided by the product family architecture. The software architecture is the top-level decomposition of a software system into its main components. In correspondence to [17], COVAMOF does not require a specific architectural description language and requires multiple views on the software architecture. The CVV captures the variability in the architecture and acts as the link between the different views.

The artefacts, e.g. source code files and executables, in the *component implementation* layer together form the software system and realize the variability provided by the features and the architecture.

### 5.3 Variation points

Variation points in the CVV are a view on the variation points in the product family, as introduced in section 2. There are five types of variation points in the CVV:

```
VPTYPES = { OPTIONAL , ALTERNATIVE ,  
           OPTIONAL VARIANT , VARIANT , VALUE }
```

- 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 that can be chosen in a predefined range.

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 artefacts, 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.

The CVV contains the following properties of the variation points:

- **Description:** A general description of the variation point.
- **Rationale:** The basis on which the software engineer should make his choice between the variants.
- **Mechanism:** Variation points in a software product family can be realized by variation points on a lower level of abstraction, or by a realization mechanism in the product family artefacts. For variation points realized by other variation points, see bullet: Realization relations. Examples of mechanisms include architectural design patterns, aggregation, inheritance, parameterization, overloading, macros, conditional compilation, and dynamically linked libraries, see also, e.g. [24], [18].
- **Binding time:** The latest lifecycle phase at which the variation point can be (re)bound, e.g. the binding time of a variation point that is implemented by conditional compilation is the compilation phase. We define the set phases as:

$$\text{PHASES} = \{ \text{PRECOMPILATION}, \text{COMPILATION}, \text{LINKING}, \text{INSTALLATION}, \text{STARTUP}, \text{RUNTIME} \}$$

- **Closing time:** Each variation point can be either *open*, i.e. new variants can be added to the variation point, or *closed*, i.e. the set of variants is fixed. Depending on the implementation mechanism, dependencies, and realization relations with other variation points, at some phase in the development lifecycle the variation point changes from open to closed. This closing time is therefore the latest lifecycle phase at which new variants can be added to the variation point. This property is not relevant for value variation points.

- **Reason:** The reason for the existence of the variation point in the product family. Examples of these reasons are to enable variable functionality, to enable variable performance or quality, or an undocumented decision in the past.

- **Scope:** The impact of the binding of the variation point on the overall system functionality or performance, between local (0) and system-wide (10).

- **Location of binding:** The location at which the variants are bound in the artefacts. Examples of these locations are a configuration file, a folder containing libraries, and a database entry.

For variation points in the CVV we define the set of possible combinations of variations points as:

$$\begin{aligned} \text{VARIANT} &= \{ (\text{V\_ID}; \text{V\_ID\_TYPE}), \\ &\quad (\text{DESCRIPTION}; \text{String}), \\ &\quad (\text{ACTIONS}; \text{String}), \\ &\quad (\text{LOCATION}; \text{String}) \} \\ \text{VARIANT\_S} &= \{ T \subseteq \prod(\text{VARIANT}) \wedge \\ &\quad \forall t \in T: \forall t' \in T: t \neq t' \Rightarrow t(\text{V\_ID}) \neq t'(\text{V\_ID}) \} \\ \text{VP} &= \{ (\text{VP\_ID}; \text{VP\_ID\_TYPE}), \\ &\quad (\text{VPTYPE}; \text{VPTYPES}), \\ &\quad (\text{DESCRIPTION}; \text{String}), \\ &\quad (\text{BINDINGTIME}; \text{PHASES}), \\ &\quad (\text{RATIONALE}; \text{String}), \\ &\quad (\text{ASPECT}; \text{String}), \\ &\quad (\text{SCOPE}; \{0,1,2,3,4,5,6,7,8,9,10\}), \\ &\quad (\text{ARTIFACT}; \text{A\_ID\_TYPE}) \} \\ \text{VP\_VAR} &= \text{VP} \cup \{ (\text{VARIANTS}; \text{VARIANT\_S}), \\ &\quad (\text{CLOSINGTIME}; \text{PHASES}) \} \\ \text{VP\_VAL} &= \text{VP} \cup \{ (\text{RANGE}; \text{P}(\mathbb{R})) \} \\ \text{VP\_S} &= \{ T \subseteq (\prod(\text{VP\_VAR}) \cup \prod(\text{VP\_VAL})) \\ &\quad \wedge \forall t \in T: \forall t' \in T: t \neq t' \Rightarrow t(\text{VP\_ID}) \neq t'(\text{VP\_ID}) \} \end{aligned}$$

A configuration of a set of variation points is defined as follows:

$$\begin{aligned} \text{CONFIG\_VAR} &= \{ (\text{VP\_ID}; \text{VP\_ID\_TYPE}), \\ &\quad (\text{VARIANTS}; \text{P}(\text{V\_ID\_TYPE})) \} \\ \text{CONFIG\_VAL} &= \{ (\text{VP\_ID}; \text{VP\_ID\_TYPE}), \\ &\quad (\text{VALUE}; \mathbb{R}) \} \\ \text{CONFIG\_S} &= \{ T \subseteq (\prod(\text{CONFIG\_VAR}) \cup \prod(\text{CONFIG\_VAL})) \\ &\quad \wedge \forall t \in T: \forall t' \in T: t \neq t' \Rightarrow t(\text{VP\_ID}) \neq t'(\text{VP\_ID}) \} \end{aligned}$$

The CVV furthermore defines two relational structures on the variation points, i.e. the artefacts and the realization relation. These relations provide a hierarchical organization and traceability in the set of variation points, respectively.

- **Artefacts:** Artefact entities in the CVV are a view on the artefacts in the product family that are relevant for product family engineering. Examples of these artefacts are features (in the features layer), architectural components (in the architecture layer), and C header files (in the component implementation layer). These artefacts and the relations in the CVV define a hierarchical organization on the variation points, in correspondence to requirement R2. The

properties of artefacts are the abstraction layer, a description, the location in the asset base, and the aggregate artefact. We therefore define:

$$\begin{aligned} \text{ARTIFACT} &= \{(\underline{\mathbf{A\_ID}}; \mathbf{A\_ID\_TYPE}), \\ &\quad (\underline{\mathbf{ABSTRACTIONLAYER}}; \mathbf{LAYERS}), \\ &\quad (\underline{\mathbf{DESCRIPTION}}; \mathbf{String}), \\ &\quad (\underline{\mathbf{LOCATION}}; \mathbf{String}), \\ &\quad (\underline{\mathbf{PARTOF}}; \mathbf{A\_ID\_TYPE})\} \\ \text{ARTIFACT\_S} &= \{T \subseteq \prod(\text{ARTIFACT}) \wedge \\ &\quad (\forall t \in T: \forall t' \in T: t \neq t' \Rightarrow t(\underline{\mathbf{A\_ID}}) \neq t'(\underline{\mathbf{A\_ID}})) \wedge \\ &\quad \{t(\underline{\mathbf{PARTOF}}) \mid t \in T\} \subseteq \{t(\underline{\mathbf{A\_ID}}) \mid t \in T\}\} \end{aligned}$$

- **Realization relations:** Variation points that have no associated realization mechanism in the product family artefacts are realized by variation points on a lower level of abstraction and therefore provide traceability in the CVV. The realization relation in the CVV defines a set of 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, and addresses requirement R5. The rules in the realization relations are of the form “IF (variant binding) THEN (variant binding)”. We therefore define:

$$\begin{aligned} \text{REALIZATION} &= \{(\underline{\mathbf{R\_ID}}; \mathbf{R\_ID\_TYPE}), \\ &\quad (\underline{\mathbf{RULE}}; \mathbf{CONFIG\_S} \rightarrow \mathbf{CONFIG\_S})\} \\ \text{REALIZATION\_S} &= \{T \subseteq \prod(\text{REALIZATION}) \wedge \\ &\quad \forall t \in T: \forall t' \in T: t \neq t' \Rightarrow t(\underline{\mathbf{R\_ID}}) \neq t'(\underline{\mathbf{R\_ID}}) \wedge \\ &\quad \forall t \in T: (\text{domain}(t(\underline{\mathbf{RULE}})) \setminus \{\underline{\mathbf{VP\_ID}}\}) \cap (\text{range}(t(\underline{\mathbf{RULE}})) \setminus \{\underline{\mathbf{VP\_ID}}\}) = \emptyset\} \end{aligned}$$

## 5.4 Dependencies

The CVV models all types of dependencies that occur in industrial product families. Dependencies are restrictions the binding of one or more variation points. Besides containing a description of each dependency, the CVV primarily distinguishes between statically and dynamically analyzable dependencies, corresponding to requirement R6. We define the possible attribute values of dependencies as:

$$\text{DEP} = \{(\underline{\mathbf{DEP\_ID}}; \mathbf{DEP\_ID\_TYPE}), (\underline{\mathbf{DESCRIPTION}}; \mathbf{String})\}$$

Simple dependencies specify the restriction on the binding of variation points that can be expressed by a Boolean expression. The validity of these dependencies can be calculated from the selection of variants of the associated variation and can therefore be verified without testing the software system. We therefore refer to these dependencies as statically analyzable dependencies. The CVV specifies a function *valid* from a configuration of the associated variation points to true or false, indicating whether the dependency is not violated and models these dependencies as:

$$\begin{aligned} \text{STATICALDEP} &= \text{DEP} \cup \{(\underline{\mathbf{VALID}}; \mathbf{CONFIG\_S} \rightarrow \{\text{true}, \text{false}\})\} \\ \text{STATICALDEP\_S} &= \{T \subseteq \prod(\text{STATICALDEP}) \wedge \\ &\quad \forall t \in T: \forall t' \in T: t \neq t' \Rightarrow t(\underline{\mathbf{DEP\_ID}}) \neq t'(\underline{\mathbf{DEP\_ID}})\} \end{aligned}$$

Experience in industrial product families showed that, in general, dependencies are more complex, cannot be written down in a formal way and typically affect a large number of variation points. Dependencies can therefore, in many cases, not be stated by such a Boolean expression. They often have a more complex or imprecise character, such as. “The combination of component A and component C might have a negative effect on the overall system performance”. The validity of the dependency therefore cannot be calculated from the variant selections. We therefore refer to these dependencies as dynamically analyzable dependencies. The CVV contains for each dynamically analyzable dependency the following properties:

- **Aspect:** Each such dependency is associated to an aspect that can be expressed by a Real ( $\mathfrak{R}$ ) value  $A$ , e.g. the performance of the system graded between 0 and 10 or the memory consumption in terms of kilobytes.
- **Valid range:** Depending on e.g. the application domain, the customer requirements, or communicating subsystems, these dependencies specify a function from  $\mathfrak{R}$  to  $\{\text{true}, \text{false}\}$ , indicating whether a value  $A$ , as mentioned above is acceptable. An example of such a function is  $f(x) = \text{if } x > 100.0 \text{ then true else false}$ .
- **Associations:** For dynamically analyzable dependencies, the CVV distinguishes between three types of associations of the 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 value of  $A$ .

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

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

*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 value of  $A$ .

```

ASSOCTYPES = {PREDICTABLE, DIRECTIONAL, UNKNOWN}
ASSOC      = { (VP_ID ; VP_ID_TYPE),
              (ASSOCTYPE ; ASSOCTYPES) }
PREDICTABLE = ASSOC ∪
              { (PREDICTION;P(V_ID) × P(V_ID) → (ℝ → ℝ) ) }
DIRECTIONAL = ASSOC ∪
              { (DIRECTION;P(V_ID) × P(V_ID) → { positive, neutral, negative} ) }
UNKNOWN     = ASSOC ∪
              { (ABSTRACT ; String) }
ASSOCIATION_S = { T ⊆ (∏(PREDICTABLE) ∪
                       ∏(DIRECTIONAL) ∪
                       ∏(UNKNOWN)) ∧
                  ∀t ∈ T: ∀t' ∈ T: t ≠ t' ⇒ t(VP_ID) ≠ t'(VP_ID) }

```

- **Test:** The check that software engineers during product derivation have to perform in order to determine the value for  $A$ . An example of such a test is the running of the complete product on a test bench.
- **Strategy:** An informal documentation which specifies a strategy to resolve the dependency during product derivation.

The set of possible combinations of dynamically analyzable dependencies is:

```

DYNAMICALDEP = DEP ∪ { (ASPECT ; String),
                      (VALIDRANGE; ℝ → {true, false} ),
                      (TEST ; String),
                      (STRATEGY ; String),
                      (ASSOCIATIONS ; ASSOCIATION_S),
DYNAMICALDEP_S = { T ⊆ ∏(DYNAMICALDEP) ∧
                  ∀t ∈ T: ∀t' ∈ T: t ≠ t' ⇒ t(DEP_ID) ≠ t'(DEP_ID)
                  }

```

In order to support with requirement R7, dependency interactions are explicitly modelled in the CVV. These dependency interactions 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. We therefore define:

```

DEPINTERACT = { (I_ID;I_ID_TYPE),
                (DEPENDENCIES;P(DEP_ID_TYPE)),
                (ORIGIN;String),
                (COPING;String) }
DEPINTERACT_S = { T ⊆ ∏(DEPINTERACT) ∧
                  ∀t ∈ T: ∀t' ∈ T: t ≠ t' ⇒ t(I_ID) ≠ t'(I_ID) }

```

## 5.5 Intrinsic variability modelling

In most existing software modelling techniques the software assets are modelled extrinsically, i.e. the model is maintained outside of the assets. An artefact with intrinsic variability is an artefact that, in domain engineering, is integrated with its own variation point model information. This information encompasses all the information regarding variability that is needed in

the domain engineering and the application engineering phase. The artefacts can therefore provide their own variability information and all the artefacts together provide the complete variability model of the software product family. As changes to the artefacts are changes to the model, the model remains consistent with the artefacts.

COVAMOF allows for the intrinsic modelling of (parts of) the CVV in order to comply with requirement R8 and therefore solves problem P7, the possible drift between the variability model and the product family artefacts.

## 5.6 Tool support for COVAMOF

Corresponding to requirement R4, we have developed Mocca. Mocca is a tool to support the management of the COVAMOF Variability View (CVV) on the level of application engineering and domain engineering. A screenshot of Mocca is presented in Figure 1.

In order to comply with requirement R3, Mocca allows for multiple views on the CVV. These views can be used to manage the CVV, each focusing on a different aspect. Currently, the Mocca implementation supports the management of the CVV from the variation point view and the dependency view:

The *Variation Point View* 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. 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.

All Mocca functionality is implemented in the Java programming language as extension to the Eclipse 3.0 Platform [12]. Eclipse allows for a tight coupling between the source code and Eclipse plug-ins. Implementing Mocca as an Eclipse plug-in therefore allows us to provide a tight coupling between the source code and the variability model of a product family.

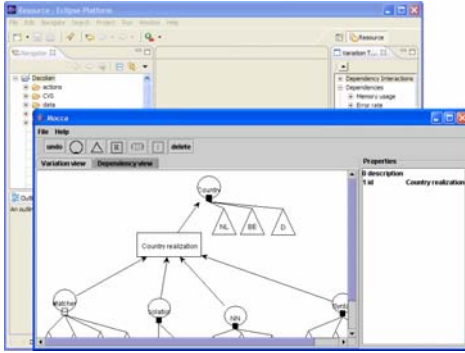


Figure 1 Screenshot of the Mocca tool suite

## 6. Validation

We have validated COVAMOF by applying the framework at the Intrada® product family of Dacolian B.V. and we are currently validating COVAMOF at the Combat Management product family at Thales Naval Netherlands. We check whether all relevant knowledge about the product family can be captured in the CVV and whether this knowledge can be used effectively during application engineering and domain engineering. In this section, we present an excerpt of the CVV of the Intrada® product family, statistical results and the experiences of applying COVAMOF.

### 6.1 The CVV of the Intrada product family

In order to illustrate the CVV we present an excerpt of the CVV of the Intrada product family at Dacolian B.V. We use the following variation points: the variant feature “Countries” (VP1), which defines the countries of which license plates can be recognized, the neural network components that are used for image recognition (VP2), the matcher modules that are used to match images with earlier recognized characters (VP3), the algorithm used to select the for recognition possible interesting parts from an image (VP4), and the algorithms used to determine the syntactical correctness of recognized license plates (VP5).

The selection of the neural network components (VP2), the selection of matcher modules (VP3) and the selection of syntax algorithms (VP5) in the architecture *realize* the selection of the supported countries (VP1) in the features.

In this excerpt, two dynamically analyzable dependencies are the memory usage of the system in kilobytes and the number of incorrectly recognized license plates in percentages. The memory usage depends on the binding of VP1, VP2 and VP3. It is known in which direction VP1 and VP2 influence the memory usage (directional association), this is not

known of VP3 (unknown association). The error rate depends on the binding of VP2, VP3 and VP4. In this case, only the directional influence of VP3 is known.

A *dependency interaction* specifies how to cope with these related dependencies. This coping depends on the target platform: if the system should run on a large mainframe, the memory usage is not the main issue and the software engineer can focus on optimizing the error rate. On a PDA, however, minimal memory usage is crucial and the software engineers cannot optimize the error rate without keeping in mind what would be the effect on the memory usage. As matcher modules do effectively decrease the error rate, but might or might not consume a lot memory, variation point VP3 plays a key role in this interaction.

### 6.2 Statistical results

The Intrada Product family contains around 7000 variation points (6872 exact). Most of these variation points are of the value type (94.6%). The remaining variation points are almost equally distributed among the other types (1.7% optional, 1.2% alternative, 1% optional variant, and 1.5% variant).

**Binding time:** Most of the value variation points have their binding time at compilation time. These parameter values are determined during off-line tests and experiments. The determined values are then included the component code (83.6%). There is a tendency to place the binding time of variation points at post deployment time (installation, start-up, and runtime). On the feature level, the configuration is controlled by six variation points that are bound at start-up time.

**Closing time:** All the non-value variation points are open before compilation time and closed during compilation time. A detailed analysis of the variation points revealed dependencies between variation points that limit the number of variants that can be associated with one variation point. This is mainly caused by how the runtime representation of the variants is implemented.

**Scope:** The parameter variation points have a severe impact on the system performance but their value has no impact on the configuration of the remaining variability. In this respect is their scope local. Typically, the alternative variation points within the Intrada product family have a system wide scope.

### 6.3 Experiences

The application of COVAMOF at Dacolian resulted in a substantial improvement of the product derivation process as well as the evolution of the product family.

The main advantage of having the CVV explicitly available is the much better overview of the variability, in particular the dynamically analyzable dependencies. The practitioners furthermore indicated that, due to the better overview of the variability, the product family artefacts have evolved in domain engineering. This resulted in a better performance of all products derived since.

## 7. Conclusion

The explicit representation of variability has been identified as a key aspect of the successful exploitation of software product families. In earlier work, we presented the problems and issues we identified in current software product family practice at several industrial product families. Based on these problems and issues, we presented eight requirements on variability modelling approaches, and showed that none of the existing approaches addresses all requirements. In this paper, we therefore presented COVAMOF, our approach to variability modelling that represents variation points as first class citizens (R1), provides the hierarchical organization of the variation by the artefact composition structure (R2), is supported by the Mocca tool suite (R3), which provides multiple views on the variability model (R4). COVAMOF furthermore defines the realization relation between variation points, which provides traceability throughout the layers of abstraction (R5). In addition, statically and dynamically analyzable dependencies are modelled as first class citizens (R6) and the dependency interactions between these dependencies are represented in the CVV (R7). Finally, the CVV can be specified in extrinsic variability models as well as intrinsically as part of the product family artefacts (R8).

## 8. References

[1] T. Asikainen, T. Soininen, T. Männistö: *A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families*, 5th Workshop on Product Family Engineering (PFE-5), Springer Verlag Lecture Notes on Computer Science Vol. 3014 (LNCS 3014), pp. 225-249, May 2004.

[2] F. Bachman, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, A. Vilbig: *Managing Variability in Product Family Development*, 5th Workshop on Product Family Engineering (PFE-5), Springer Verlag Lecture Notes on Computer Science Vol. 3014 (LNCS 3014), pp. 66-80, May 2004.

[3] M. Becker: *Mapping Variability's onto Product Family Assets*, Proceedings of the International Colloquium of the

Sonderforschungsbereich 501, University of Kaiserslautern, Germany, March 2003.

- [4] J. Bosch: *Design & Use of Software Architectures, Adopting and Evolving a product-line approach*, Addison-Wesley, ISBN 0-201-67494-7, 2000.
- [5] 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.
- [6] M. Clauss: *Modeling variability with UML*, GCSE 2001 - Young Researchers Workshop, September 2001.
- [7] P. Clements, L. Northrop: *Software Product Lines: Practices and Patterns*, SEI Series in Software Engineering, Addison-Wesley, ISBN: 0-201-70332-7, 2001.
- [8] The ConIPF project (Configuration of Industrial Product Families), <http://segroup.cs.rug.nl/conipf>.
- [9] S. Deelstra, M. Sinnema, J. Bosch: *Product Derivation in Software Product Families; A Case Study*, accepted for the Journal of Systems and Software, 2003.
- [10] S. Deelstra, M. Sinnema, J. Bosch: *A Product Derivation Framework for Software Product Families*, 5th Workshop on Product Family Engineering (PFE-5), Springer Verlag Lecture Notes on Computer Science Vol. 3014 (LNCS 3014), pp. 473-484, May 2004.
- [11] S. Deelstra, M. Sinnema, J. Nijhuis, J. Bosch, *COSVAM: A Technique for Assessing Software Variability in Software Product Families*, accepted for the 20th IEEE International Conference on Software Maintenance (ICSM 2004), September 2004.
- [12] Eclipse Platform, <http://www.eclipse.org>.
- [13] H. Goma, 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.
- [14] J. van Gorp, J. Bosch: *Design Erosion: Problems & Causes*, Journal of Systems and Software, 61(2), pp. 105-119, March 2002.
- [15] C. Hofmeister, R. Nord, D. Soni: *Applied Software Architecture*, Addison-Wesley, 2000.
- [16] L. Hotz, T. Krebs: *Supporting the Product Derivation Process with a Knowledge-based Approach*, Software Variability Management Workshop at ICSE 2003, 2003.
- [17] IEEE Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE Standard P1471), IEEE Architecture Working Group (AWG), 2000.

- [18] I. Jacobson, M. Griss, P. Jonsson: *Software Reuse. Architecture, Process and Organization for Business Success*. Addison-Wesley, ISBN: 0-201-92476-5, 1997.
- [19] M. Jaring and J. Bosch: *Variability Dependencies in Product Family Engineering*, 5th Workshop on Product Family Engineering (PFE-5), Springer Verlag Lecture Notes on Computer Science Vol. 3014 (LNCS 3014), pp. 81-97, May 2004.
- [20] R. van Ommering: *Building Product Populations with Software Components*, in Proceedings of the 24th International Conference on Software Engineering (ICSE'02), May 2002.
- [21] 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.
- [22] D. Parnas, *On the Design and Development of Program Families*, IEEE Transactions on Software Engineering, SE-2(1):1-9, 1976.
- [23] M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch: *COVAMOF: A Framework for Modeling Variability in Software Product Families*, Proceedings of the Third Software Product Line Conference (SPLC 2004), Springer Verlag Lecture Notes on Computer Science Vol. 3154 (LNCS 3154), pp. 197-213, August 2004
- [24] M. Svahnberg, J. Gorp, J. Bosch: *A Taxonomy of Variability Realization Techniques*, technical paper ISSN: 1103-1581, Blekinge Institute of Technology, Sweden, 2002.
- [25] 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.