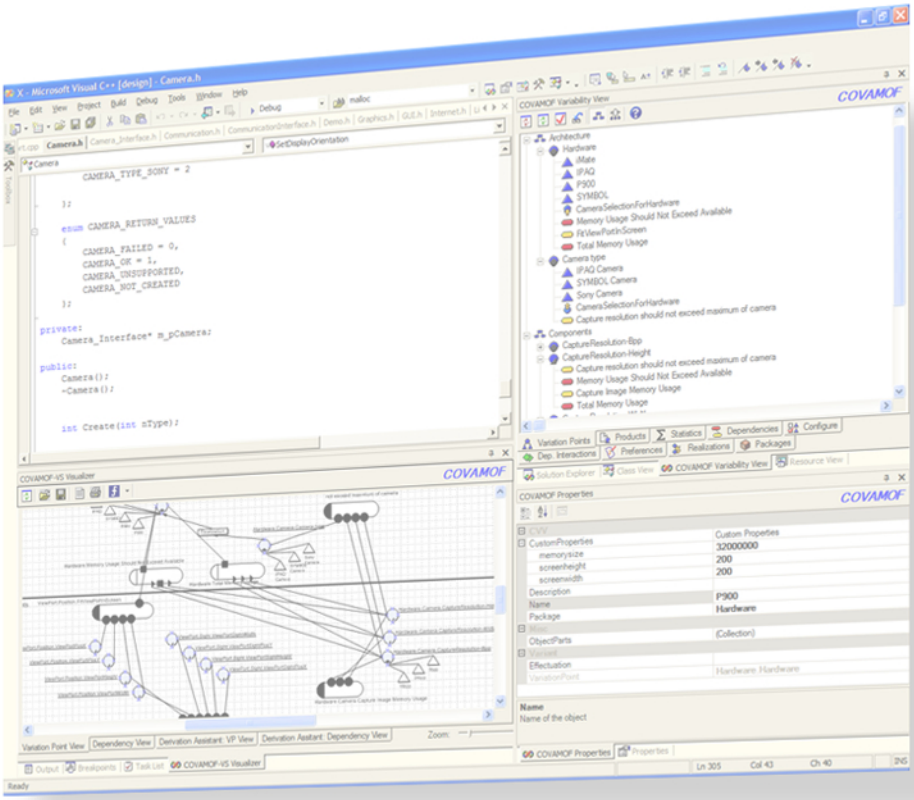
MANAGING THE COMPLEXITY OF VARIABILITY IN SOFTWARE PRODUCT FAMILIES



```
        CAMERA_TYPE_SONY = 2

    };

    enum CAMERA_RETURN_VALUES
    {
        CAMERA_FAILED = 0,
        CAMERA_OK = 1,
        CAMERA_UNSUPPORTED,
        CAMERA_NOT_CREATED
    };

private:
    Camera_Interface* m_pCamera;

public:
    Camera();
    ~Camera();


    int Create(int nType);
```

Architecture
- Hardware
  - iMate
  - IPAQ
  - P900
  - SYMBOL
  - CameraSelectionForHardware
  - Memory Usage Should Not Exceed Available
  - FitViewPortInScreen
  - Total Memory Usage
- Camera type
  - IPAQ Camera
  - SYMBOL Camera
  - Sony Camera
  - CameraSelectionForHardware
  - Capture resolution should not exceed maximum of camera

Components
- CaptureResolution-Bpp
- CaptureResolution-Height
  - Capture resolution should not exceed maximum of camera
  - Memory Usage Should Not Exceed Available
  - Capture Image Memory Usage
  - Total Memory Usage

CVV	
CustomProperties	Custom Properties
memorysize	32000000
screenheight	200
screenwidth	200
Description	
Name	P900
Package	Hardware
Misc	
ObjectParts	(Collection)
Variant	
Effectuation	
VariationPoint	Hardware Hardware

Name
Name of the object

Sybren Deelstra & Marco Sinnema

RIJKSUNIVERSITEIT GRONINGEN

Managing the Complexity of Variability in Software Product Families

Proefschrift

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. F. Zwarts,
in het openbaar te verdedigen op
vrijdag 9 mei 2008
om 14.45 uur

door

Sijbren Keimpe Deelstra
geboren op 12 oktober 1979
te Drachten

en om
16.15 uur

door

Marco Sinnema
geboren op 7 januari 1978
te Drachten

Samenvatting

Sinds de zestiger jaren wordt algemeen aangenomen, dat het hergebruik van software een goede oplossing is voor de problemen met betrekking tot de kosten, kwaliteit en levertijd van software producten. Vanaf de jaren negentig vormen software productfamilies een belangrijke toevoeging aan de bestaande methodes om het hergebruik van software te realiseren. Het idee achter productfamilies is, om binnen de gehele organisatie het gebruik van de verschillen en overeenkomsten tussen de te ontwikkelen producten expliciet te plannen. De invoering van deze aanpak in de industrie heeft de voordelen al naar voren gebracht. Het afleiden van deze producten is echter nog te duur, kost teveel tijd en er zijn teveel experts bij nodig. De uitdaging is daarom om de kosten voor het afleiden van deze producten te verlagen.

Het doel van het onderzoek, dat we in ons proefschrift presenteren, is om technieken te ontwikkelen en te valideren die de hoge kosten van het afleiden van producten verlagen. Omdat het daarbij belangrijk is dat deze technieken bruikbaar zijn in de industrie, begint het proefschrift met een studie van de productafleiding in industriële toepassingen. We bieden een raamwerk van concepten die deze productafleiding beschrijft en binnen dit raamwerk presenteren wij onze analyse van de praktijk. Verder geven we een overzicht van de onderliggende oorzaken van de hoge kosten van het afleiden van deze producten.

Het expliciet maken van de variabiliteit in een variabiliteitsmodel pakt veel van die onderliggende oorzaken aan. Daarom vergelijken en classificeren we bestaande technieken voor variabiliteitsmodellering, die beschreven zijn in de literatuur, op basis van de resultaten van onze analyse van de praktijk. Binnen deze classificatie analyseren we de modelleringsvormen en de tools *die bij de technieken horen. Verder identificeren we de overeenkomsten en verschillen tussen de technieken en benoemen we de nog op te lossen problemen.*

Tenslotte beschrijft het proefschrift COVAMOF, onze methode voor variabiliteitsbeheersing. COVAMOF pakt zowel de belangrijkste uitdagingen binnen productafleiding als de nog op te lossen problemen van de bestaande technieken aan. De kern van COVAMOF bestaat uit een modelleringstaal en een uitgebreide toolset. Bovenop deze kern hebben we een afleidingsproces en een assessmentmethode voor productfamilies ontwikkeld. Het afleidingsproces beschrijft, hoe de modelleringstaal en de tools kunnen worden gebruikt om producten efficiënt van de productfamilie af te leiden. Het doel van de assessmentmethode is om de variabiliteit binnen de productfamilie te beoordelen op haar geschiktheid voor bestaande en toekomstige producten. In dit laatste deel van het proefschrift beschrijven we het afleidingsproces en de assessmentmethode in detail en valideren we COVAMOF met behulp van de resultaten van een experiment in de industrie. Deze resultaten tonen aan, dat COVAMOF de kosten van productafleiding reduceert, zowel wat betreft de benodigde inspanning, de afhankelijkheid van experts, als de levertijd.

Abstract

Since the 1960s, reuse has been the long-standing notion to solve the cost, quality and time-to-market issues associated with development of software applications. A major addition to existing reuse approaches since the 1990s are software product families. Although the adoption of this approach in industry clearly showed its benefits, several major challenges remain. One of these main challenges is reducing the high cost of product derivation; it is expensive, takes too long, and requires too much involvement of experts.

The goal of the work we present in this thesis is to define and validate techniques that are useful in industry, and that reduce the high cost of product derivation. To this extent, this thesis starts with a study on product derivation in practice. It provides (1) a framework of concepts that explains product derivation, (2) a description of case studies in terms of this framework, and (3) the identification of the underlying causes of the high cost of product derivation.

Making variability explicit in a model addresses a large number of the underlying causes. Based on the results of the product derivation study, this thesis therefore compares and classifies variability modeling techniques that are found in literature. This classification analyses the modeling concepts and tools, and identifies similarities, differences, and open issues.

Finally, this thesis describes our variability management framework COVAMOF. This framework addresses the main challenges of product derivation, as well as the open issues of variability modeling techniques. The core of COVAMOF consists of a variability modeling language and a tool-suite. On top of this core, we defined a derivation process and a *variability assessment technique. In the last parts of this thesis, we illustrate and validate COVAMOF by presenting the results of the application of COVAMOF in industry. These results show that COVAMOF reduces product derivation cost in terms of expert dependency, effort, and time-to-market.*

Acknowledgements

Several people provided valuable contributions to the realization of this thesis. First of all, we would like to thank our promotor, prof. dr. ir. Jan Bosch, for giving us the opportunity to work closely with industry, with the freedom to pursue our own goals. We particularly enjoyed the result-driven atmosphere, which let to many peer-reviewed publications of our work.

We would like to thank our co-promotor, dr. ir. Jos Nijhuis, for the insightful discussions, his pragmatic attitude towards research, and the fun of sharing the same office.

We also express our gratitute towards the reading committee, prof. dr. Kyo Kang, prof. dr. ir. Philippe Kruchten, and prof. dr. Klaus Schmid, for their detailed and committed analysis of our thesis.

Finally, we like to thank several organizations and projects that contributed to this thesis, i.e. the EU and ConIPF for funding part of our research under contract no. IST-2001-34438, Thales Nederland, Robert Bosch GmbH, and Dacolian B.V. for participating in our case studies, and Océ Technologies B.V. for printing this thesis.

This thesis was printed on a VarioPrint 6250

Printing for Professionals

Table of Contents

Introduction

Chapter 1 This thesis

The collaborative research of Marco Sinnema and Sybren Deelstra has reached an important milestone. The past years have been both fun and hard work, but now is the time to unify the result of these years into a single thesis. In this chapter, we discuss the background of this research. We start with a brief introduction to software development in general, and present the motivation for our research. We furthermore present our research questions and the methodologies we applied to answer them. We finalize this chapter by discussing how we distributed the responsibilities for different topics in our collaborative research over the two of us.

1.1. The Software Age and This Thesis

Software made a tremendous contribution to the technical advancement of humanity. This has led to a situation where software plays a growing and central role in nearly every aspect of our lives (ACM and IEEE, 1999). Nowadays, almost anything with a wire or batteries contains software. Looking at all major fields, the list of examples seems endless; Software is used in *communication* (e.g. phones, the internet), *transportation* (e.g. parking space detection, car navigation systems, traffic flow control, speed enforcement), *healthcare* (e.g. medical scanner, patient data, surgical instruments), *entertainment* (e.g. home cinema, TV broadcasts, videogames), *recreation* (e.g. score boards, challenge ladders, identification and registration systems), *law-enforcement* (e.g. illegal content detection, incident report systems), *finance* (e.g. stock exchange, ATMs, accounting software), *construction* (e.g. project management, design, structural analysis), and *manufacturing* (e.g. inventory, product life-cycle management, CAD/CAM for 3D machining, mold making, or reverse engineering). Instead of the Information Age, we might thus just as well call our century the Software Age.

To develop this software, organizations take up the challenge of an optimization problem. This optimization problem involves balancing the maximization of results (e.g. short time-to-market, perfect functionality and quality, profit) on the one hand, with the minimization of required resources (e.g. people, time, investment) on the other hand. That this optimization problem really *is* a challenge has already been shown by examples from the past (see Example 1).

***Example 1**. Historical examples where software development proved to be a challenging task:*

- ***Schedule and budget overruns.** In 1987, California's DMV (Department of Motor Vehicles) decides to merge the state's driver and vehicle registration systems. Along the way, it sees the projected cost explode to 6.5 times the expected price and the delivery date recede to 1998. After seven years, the project was canceled after having received a total of $44.3-million (Gibbs, 1994)*
- ***Damage.** Well-known examples of software errors that cause physical and economical damage (and as a result probably some emotional too) are those related to the space agencies. A calculation error in the Ariane-5 rocket, for example, caused both primary and backup inertial reference systems to fail (Lions, 1996). As a result, the rocket changed it course, and automatically self-destructed after 39 seconds of flight. Total cost: $500 million payload, excluding launch cost, and 2-3 years delay in the project.*
- ***Deaths:** A striking example of fatal injuries due to software errors is the infamous Therac-25, which, amongst others, overdosed two patients at the East Texas Cancer Center, March 1986. One patient died after suffering nausea, vomiting, paralysis of his left vocal cord (which left him unable to speak), left arm and both legs, a neurogenic bowel and bladder, skin infections, and so on. He died from complications of the overdose five months after the accident. The other patient died three weeks after the accident. He had disorientation that progressed to coma, fever to 104 degrees Fahrenheit, and neurological damage (Leveson and Turner, 1993).*

As the examples above show, ineffective software development can cause schedule and budget overruns, economical, physical and emotional damage, and even deaths. Techniques that help to deal with this challenge are thus a valuable contribution to society. To be able to develop those techniques, the number one question to be answered is what actually causes these software development problems. Brooks (1986) attributes the existence of software problems in general to the inherent complexity of software, which in turn is the result of “the complexity of the problem domain, the difficulty of managing the development process, the flexibility possible through software, and the problems of characterizing the behavior of discrete systems” (Booch, 1991).

The interesting cause of software development issues in the context of this thesis is *complexity*. The trend over the past decades has been that the complexity of software systems continues to grow. As IBM notes in Autonomic computing - IBM’s Perspective on the State of Information Technology:

***Example 2**. Autonomic computing - IBM's Perspective on the State of Information Technology (IBM, 2001):*

"Follow the evolution of computers from single machines to modular systems to personal computers networked with larger machines and an unmistakable pattern emerges: incredible progress in almost every aspect of computing—microprocessor power up by a factor of 10,000, storage capacity by a factor of 45,000, communication speeds by a factor of 1,000,000—but at a price. Along with that growth has come increasingly sophisticated architectures governed by software whose complexity now routinely demands tens of millions of lines of code. Some operating environments weigh in at over 30 million lines of code created by over 4,000 programmers!

The Internet adds yet another layer of complexity by allowing us to connect—some might say entangle—this world of computers and computing systems with telecommunications networks. In the process, the systems have become increasingly difficult to manage and, ultimately, to use—ask anyone who's tried to merge two I/T systems built on different platforms, or consumers who've tried to install or troubleshoot DSL service on their own. In fact, the growing complexity of the I/T infrastructure threatens to undermine the very benefits information technology aims to provide. Up until now, we've relied mainly on human intervention and administration to manage this complexity. Unfortunately, we are starting to gunk up the works.

Consider this: at current rates of expansion, there will not be enough skilled I/T people to keep the world's computing systems running. Unfilled I/T jobs in the United States alone number in the hundreds of thousands. Even in uncertain economic times, demand for skilled I/T workers is expected to increase by over 100 percent in the next six years. Some estimates for the number of I/T workers required globally to support a billion people and millions of businesses connected via the Internet—a situation we could reach in the next decade—put it at over 200 million, or close to the population of the entire United States.

Even if we could somehow come up with enough skilled people, the complexity is growing beyond human ability to manage it. As computing evolves, the overlapping connections, dependencies, and interacting applications call for administrative decision-making and responses faster than any human can deliver. Pinpointing root causes of failures becomes more difficult, while finding ways of increasing system efficiency generates problems with more variables than any human can hope to solve".

As you may have noticed in the example above, IBM primarily focuses on complexity from the viewpoint of users and administrators of the systems. In the report, they propose a paradoxal approach to solve this problem: make user interaction and administration simpler, by making the systems more complex (through autonomic computing). This does thus not reduce software development problems, however. On the contrary, if not done carefully, autonomic computing could make software development even more problematic.

In this thesis, we focus on the complexity of software systems from the viewpoint of software development. As the title of this thesis already suggests, we deal with a specific type of software complexity, i.e. the complexity of software variability. Below, we explain the term software variability, and motivate the selection of this particular topic.

1.2. Motivation

"Any customer can have a car painted any colour that he wants so long as it is black" - *Henry Ford.*

Since the industrial revolution at the time of Ford (1900s), a lot has changed. The technological advances, and a shift in society to be more focused on the individual, made that 'any color' is not just 'black' anymore. Product diversification allows targeting niche markets and creates value as it addresses the *diversity of taste* and the *taste for diversity* of (groups of) customers (see, for example, Dixit and Stiglitz (1977)). Diversity of taste means that customers like different sets of features (functionality and quality). This liking, however, also creates the necessity to support different features. For example, modern software systems are no isolated entities anymore, but are highly interconnected with other systems. Customers therefore not only like different features for new systems, but they require these systems to operate in environments that differ per customer as well.

Taste for diversity, on the other hand, is related to the ability to self-express and the feeling of identity. Well-known and recent examples of addressing this taste are the ability to personalize clothing, phone cases, credit cards, and personalized marketing campaigns. This taste for diversity plays a similar role in software systems; for example, with personalized user interfaces.

The consequence of diversity is that organizations end up simultaneously developing multiple software systems, and that these systems have similarities and differences in their features. These similarities pose an opportunity to address the optimization problem we mentioned in the previous section; explicitly planning the reuse of sections of code that deal with these similarities offers the ability to substantially decrease cost, and time-to-market, and increase the quality. The intra-organizational reuse through the explicitly planned exploitation of similarities between related products is called a software product family (Bosch, 2000;

Clements and Northrop, 2001; van der Linden et al., 2007; Jazayeri et al., 2000; Weiss and Lai, 1999).

In a software product family, software products are developed in a two-stage process, i.e. a domain engineering stage and a concurrently running application engineering stage (Linden, 2002). Domain engineering involves, amongst others, identifying commonalities and differences between product family members and implementing a set of reusable software artifacts (e.g. components or classes). It takes care that commonalities can be exploited economically, while at the same time the ability to vary the products is preserved. During application engineering individual products are derived from the product family, viz. constructed using a subset of the reusable software artifacts.

Why this two-stage process works is best explained with a simplified example:

Example 3. *Simplified example that shows the benefit of product family engineering:*

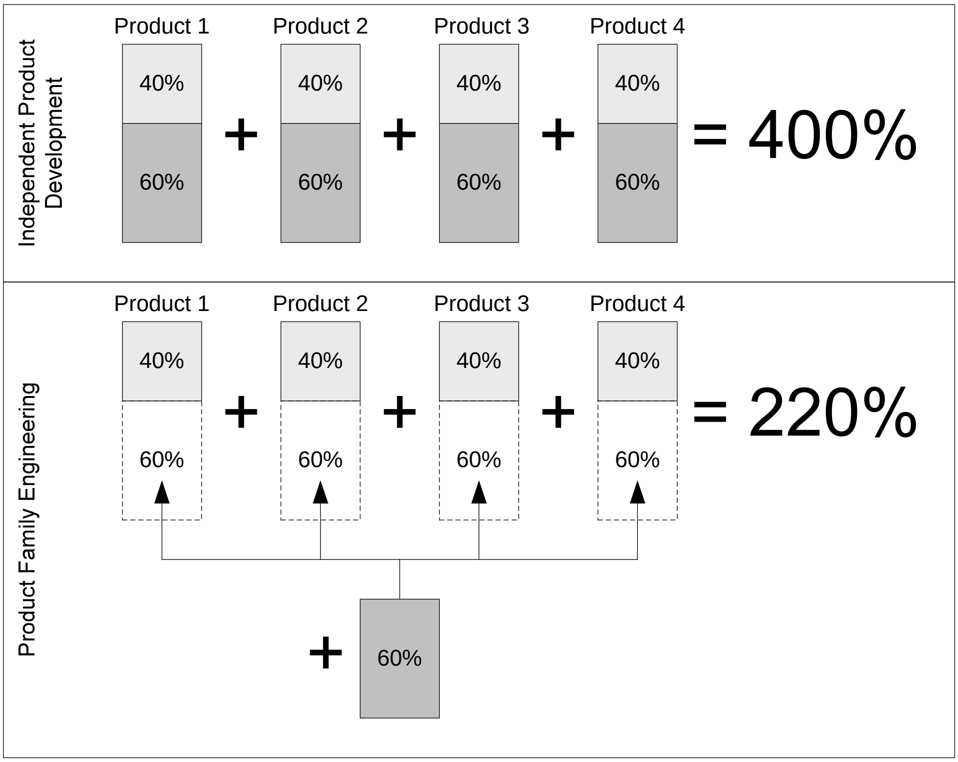


Figure 1. A simplified example of the cost savings potential of product families. This figures shows that total development cost can be decreased dramatically if we combine effort that is spent on common features into reusable code.

Let's consider a fictional organization that develops four products, say, different types of printers. These products have similarities; they all have to speak the common printing protocols such as LPR, and have the ability to translate documents described by printer description languages (PDLs) to bitmaps. But they also have differences. Some feature a scanner, one features a fax. Some are connected to legacy systems and have to support AppleTalk, while another is state of the art and supports new protocols such as IPP.

If these products would be developed independently, development cost could be distributed such as depicted in Figure 1: 60% of the cost for each product is spent on the same features, while 40% are spent on their differences. Ideally, if the organization could develop some of the common features only once (during domain engineering), the total cost for developing the four products (during application engineering) could be dramatically be reduced (see Figure 1).

As an added bonus, subsequent products may be developed faster (since we do not need to develop the common parts anymore), and the quality of the reusable code may be higher since it has been used in so many products before.

Example 3 nicely illustrates that cost for developing common features are distributed over multiple products. Calculating the cost according to that simplified example, however, would also be a simplification. In practice, there is more cost that has to be taken into account when adopting a product family based approach. As part of a model for product line economics, named SIMPLE (Structured Intuitive Model for Product Line Economics), Clements et al. (2005) created a product family cost function that does incorporate those additional cost. This function defines that the cost of building a single product line is the sum of:

- the organizational cost to adopt a product line approach (C_{org}) ,
- the cost to develop the product family artifacts during domain engineering (C_{cab}) ,
- and the cost to derive all products from the product family artifacts during application engineering. These cost break down into the cost to use the product family artifacts ($C_{reuse}(product)$) and the cost to develop the unique parts of a product ($C_{unique}(product)$).

In other words:

(Equation 1) $C_{total} = C_{org} + C_{cab} + \sum(C_{unique}(product_i) + C_{reuse}(product_i))$

Even with this additional cost, practice has shown that this philosophy works: it has proven to decrease costs and time-to-market, and increase the quality of complex software products (Linden, 2002). Since the 1990s, the product family approach has therefore been adopted by a wide variety of organizations (Bosch et al., 2001).

This does not mean, however, that there is no room for improvement. The adoption of the product family philosophy not only proved to substantially decrease the total costs, it also showed that the cost to build products from product family artifacts ($\sum(C_{unique}(product_i) + C_{reuse}(product_i))$ were much higher than expected. In 2003, for example, we (Sybren Deelstra and Marco Sinnema) were involved in an EU project called Configuration in Industrial Product Families (ConIPF, 2003). ConIPF consisted of two industrial partners (Robert Bosch GmbH and Thales Nederland B.V.) and two academic partners (Rijksuniversiteit Groningen and The University of Hamburg). It was started because – like many of the organizations our research group collaborated with – both industrial organizations felt their cost to derive products ($\sum(C_{unique}(product_i) + C_{reuse}(product_i)))$ was too high, despite their efforts in setting up a product family ($C_{org} + C_{cab}$).

This motivation from industry is also the starting point for the research we present in this thesis. While product families primarily focus on reuse, and thus commonalities between products, our research specifically focuses on the *differences* between products. The ability to derive products with differences is enabled through variability. Variability is the ability to be changed, customized, configured, or extended for use in a specific context.

On the one hand, variability is enabled through variation points, i.e. the locations in the software that enable choice at different abstraction layers in the product family artifacts. Each variation point is associated with a number of options to choose from (called variants). On the other hand, the possible configurations are restricted due to dependencies that exist between variants, and the constraints that are imposed upon these dependencies.

In our research, we were primarily interested in how these variation points, variants and dependencies affect the cost to derive products, and whether there are techniques that help to decrease those cost (variability management techniques).Our research on variability management was guided by the following research questions.

1.3. Research questions

As we discussed above, the main research question involves the reduction of application engineering effort by using these variability management techniques:

Q Overall: Which variability management techniques can effectively decrease the application engineering cost in software product families?

The answer to this question should be a useful and industrially applicable solution. Useful means that it should address the most important issues that are currently experienced by industry, i.e. the ones where solutions have a large impact on the reduction of the total product development cost (C_{total} from Equation 1). Industrially applicable means that the solutions should not only work on small-scale theoretical examples, but also on medium to large-scale systems that are used in practice.

As you may have noticed, the scope of this overall question is too large to be able to answer the question at once; we first have to gain a detailed understanding of the issues, before thinking about solutions. To answer the overall question, we therefore formulated five research questions.

1.3.1. Investigation

The first two of these research questions involve the investigation of current product derivation practice and the problems and issues related to high application engineering cost:

> ***Q1 Overview: What terminology, tools and processes are used for product derivation in research and industry?***
>
> ***Q2 Investigation: Which problems and issues do organizations face during product derivation?***

In answering this question, we did not only identify problems that are visible on the surface, but also thoroughly analyzed the underling causes and issues. Especially, we investigated the influence of these issues on the factors that are involved in the cost to reuse the product family artifacts (C_{reuse} from Equation 1). As a result, we identified two core issues that cause the total product family cost (C_{total} from Equation 1) to be higher than expected: the variability in product families is very complex and contains a large amount of implicit properties (i.e. dependencies and characteristics of the software system unknown to a most or all engineers). There are two strategies to address these issues, i.e. by *effectively dealing with complexity and implicit properties* as is, or by *effectively reducing the complexity and implicit properties* in a product family.

1.3.2. Strategy 1: Effectively dealing with complexity and implicit properties

During application engineering, engineers lack the means to effectively deal with the complexity and implicit properties. This makes it harder to reuse components that are developed in domain engineering. The complexity and implicit properties thus cause an increased cost to reuse product family artifacts (see Equation 1).

A technique that addresses this issue is variability modeling. A variability model explicitly describes the choices and dependencies that build up the variability of a product family. As we will show in this thesis, several variability modeling techniques have been developed. The publications regarding these techniques, however, were written from different viewpoints, use different examples, and rely on a different technical background. The third research question therefore involves the identification and classification of these techniques.

Q3 Variability Modeling Classification: How do the variability modeling techniques that have been proposed over the past years support product derivation?

We will show that these variability modeling techniques only model formalized knowledge and hardly address the high complexity of the provided variability. We therefore identify the need for new and improved modeling facilities, primarily for the support of the complexity and different types of knowledge. This translates into the following research question.

Q4 Variability Modeling Technique: What are key ingredients for a variability modeling technique that does address the product derivation issues of complexity and implicit properties?

- ***Which concepts are involved in handling complex dependencies and different types of knowledge?***
- ***How should these concepts be addressed in a variability modeling language?***
- ***How should such a modeling language be used during product derivation?***

1.3.3. Strategy 2: Effectively reducing complexity and implicit properties

As said, the questions above mainly address the reduction of the cost to reuse product family artifacts (C_{reuse} from Equation 1) from an application engineering point of view. They focus on a technique that makes it easier to deal with the complexity of the variability that is provided by the reuse components, i.e. variability modeling. Another way to address these issues is by reducing the complexity of the variability during domain engineering.

The investigation of the product derivation issues showed that the complexity of variability is caused by a mismatch between the variability that is provided by product families, and the variability that is required by the products. These mismatches lead to a situation where C_{reuse} or C_{unique} are higher than necessary. Our last research question is therefore related to a technique that can be used during domain engineering, i.e. variability assessment.

Q5 Variability Assessment Technique: Is it possible to assess variability so that the mismatch between the provided variability of the product family and required variability of the products can be reduced?

- ***What is variability assessment and what are the issues?***
- ***What can we do to address those issues?***

1.4. Research methodology

The goal of our research was to develop variability management techniques that are useful and industrially applicable. The applicability in industry implied a strong cooperation with this industry throughout the course of our research. To that extent, we employed the following research methodology to answer the research questions we presented in the section above.

First, we investigated the current practice of product derivation. We analyzed this current practice with respect to our view on product derivation and existing publications on this subject. Based on this analysis, we developed an initial generalized methodology that could help the industry to improve their product derivation practice.

Second, in multiple refinement iterations, we implemented this methodology into three industrial organizations. In close cooperation with the people involved, we observed the changes in their derivation practice. Based on a thorough analysis of these changes, we improved our methodology. We subsequently implemented this improved methodology in industry again, initiating the following iteration in the refinement cycle. This strategy is an instantiation of *action research* (Robson, 2002) and is visualized in Figure 2.

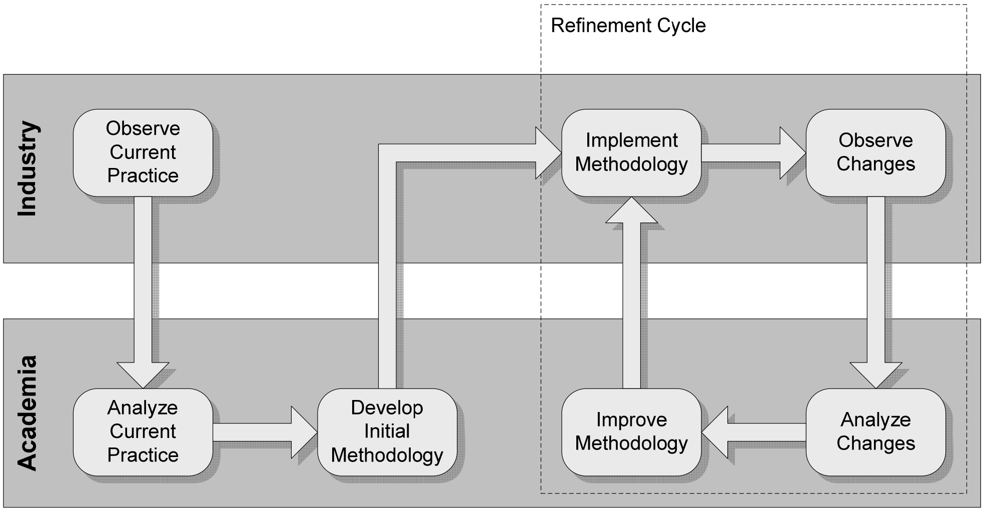


Figure 2. Research Methodology. This figure shows our strategy to develop a solution that is industrially applicable.

Over the course of employing this strategy in our research, we used three of the instruments that are available for empirical research. These instruments are the case study, the literature review, and the experiment:

- **Case Study:** A research strategy where the focus is on a case or a small set of cases (person, group, setting, organization, etc.), and which takes the context of the case into account. It involves multiple methods of data collection (e.g. structured interviews, content analysis, data archives), which result in both quantitative and qualitative data (Robson, 2002). A major benefit of a case study is its flexibility; it allows an in-depth analysis without elaborate pre-existing theoretical ideas and assumptions (in other words exploratory).
- **Literature review:** The purpose of a literature review is to summarize, classify, and compare prior research studies, reviews of literature, and theoretical articles. It is thus used to explore a particular subject, as well as to evaluate existing methods (Robson, 2002).
- **Experiment:** The experiment is a research strategy involving the assignment of participants to different conditions, manipulation of one or more variables, measuring the effect of manipulations, and the control over other variables (Robson, 2002). It is a quantitative research strategy, where the design is pre-specified. It can be used as exploratory research (to see what happens), as well as evaluation research (to see whether what happens is what you expect).

Each research question has its own suitable instrument to provide answers. We depicted the relation between these research questions and the instruments in the table below.

Table 1. The instruments we applied to answer our research questions

Research Question	Instrument	Rationale
Q1 Overview	Case study	Q1 involved exploratory research, where the direction of the outcome was unknown. It also required a large amount of detail, which is why the case study was the suitable instrument.
Q2 Investigation	Case study	Q2 also involved exploratory research, where the direction of the outcome was unknown. It also required a large amount of detail, which is why the case study was the suitable instrument.
Q3 Variability modeling classification	Literature review	The suitability of the instrument was implied by the formulation of the research question, i.e. an evaluation of existing methods.
Q4 Variability Modeling Technique	Experiment	This question involves an evaluation. As the final iteration in our refinement cycle, we wanted to validate that our variability modeling technique helps to reduce product derivation cost. It therefore implied an experiment that measures the quantitative effect of using our variability modeling technique.
Q5 Variability Assessment Technique	Literature review Case study	We wanted to identify existing approaches to variability assessment, which implied the literature review. After multiple iterations in the refinement cycle, we also wanted to perform a final evaluation of our variability assessment technique. The nature of this technique is that it guides experts in evolving a product family. Our first goal was therefore to perform a qualitative measurement, i.e. an evaluation of experiences of users of the methodology. This meant the case study was most suited.

1.5. A summary of results

While we applied the research methodologies on our research questions, we identified, developed and validated a number of results that we discuss in this thesis. A part of these results covers what we refer to as our variability management framework COVAMOF. The COVAMOF framework encompasses all solutions that we provide to answer research questions Q4 en Q5. The overview of results for all research questions is listed in the table below (Table 2).

Table 2. Results of answering the research questions.

Research Question	Result	Description
Q1 Overview	Product Derivation Framework	This product derivation framework identifies relevant terminology, a classification of product families, and a generic product derivation process.
Q2 Investigation	Problems and issues	A collection of problems and issues related to variability management, including examples, consequences, solutions and research issues.
Q3 Variability Modeling Classification	Classification of variability modeling techniques	A classification of variability modeling techniques that shows the differences, commonalities, and open issues.
Q4 Variability Modeling Technique	Conceptual view on variability	A conceptual view on software variability.
	COVAMOF Language	Our variability modeling language. This language is the instantiation of the conceptual view on variability.
	COVAMOF Tool-suite	A collection of tools that allows engineers to use the COVAMOF Language
	COVAMOF Derivation Process	A derivation process that describes how products can be derived using the COVAMOF language and tool-suite.
Q5 Variability Assessment Technique	COVAMOF Software Variability Assessment Method (COSVAM)	Our variability assessment method, a technique that structures the process to determine whether, how, and when variability should evolve.

As we said in the start of this chapter, these results are the product of the research collaboration between Marco Sinnema and Sybren Deelstra. We both had a distinct responsibility in creating these results however. We discuss this division of responsibilities below.

1.6. Responsibilities

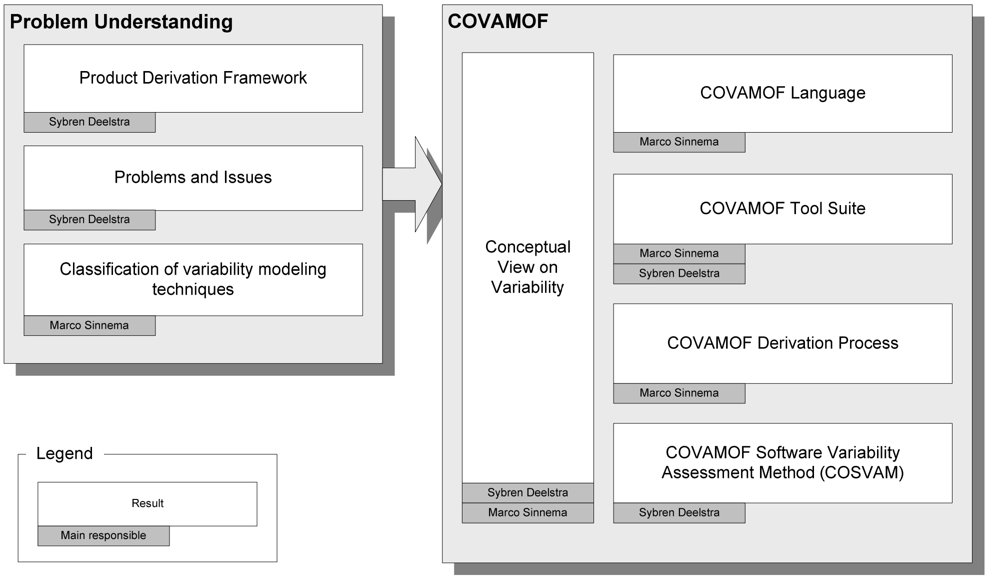


Figure 3. Main responsibilities. This figure depicts the link between results in Table 2 and the division of responsibilities.

When we started our research, we were both interested in the same field, variability management, but in slightly different areas. While Marco Sinnema's experience in dealing with languages such as XML during his former job made him most suitable for investigating modeling aspects of variability management, Sybren Deelstra's M. Sc. Thesis on evolution in software product families made him most suitable for focusing on evolutionary aspects of variability management.

We soon discovered that our work was much more related than initially planned. First, we both felt we needed a much deeper understanding of variability, especially with respect to the problems and issues that play when using variability during application engineering. This is one of the aspects where our research approach closely matched with the ConIPF project. The ConIPF project aimed to define and validate product derivation methodologies that are practicable in industrial application (ConIPF, 2003). It contained an investigation phase, where we got the opportunity to determine and understand the problems that the industrial partners faced during product derivation.

However, the commonalities between our work went further. What we realized during the investigation phase was that in order to address the variability management issues we came across, we needed more than single, independent solutions. For example, in order to address the complexity and implicit knowledge issues during application engineering, we needed new ways to make variability explicit, which was Marco's area of expertise. This new variability modeling language should be able to handle the incompleteness and imprecision of knowledge. We furthermore identified that a variability assessment method was needed that would assist architects during domain engineering. This variability assessment method should structure the process of determining how variability should evolve in response to changes in business and technology. Evolution was Sybren's area of expertise.

The assessment method, however, not only required that we both needed to develop a conceptual view of how variability manifests itself, but also required the same ways to make variability explicit. The only sensible way of tackling these issues was therefore to collaborate. This collaboration resulted in a collection of methods that grew out to be what we call the variability management framework COVAMOF. COVAMOF is one of the main results we present in this thesis (see also Section 1.5).

Due to the intense collaboration, we are both equally suited to defend this thesis. Yet, each author was the main responsible for different aspects of the results we present here. This responsibility was partly based on practicalities (Sybren started the investigation phase a few months earlier than Marco), but mainly on the experience of the authors we mentioned in the start of this section.

We depict the link between the results we listed in Table 2 and the division of responsibilities in Figure 3. As shown, Sybren was the main responsible for the start of the investigation phase, i.e. the product derivation framework, and the identification of problems and issues, while Marco was the main responsible for the subsequent classification of variability modeling techniques. This classification determines how other techniques approach the variability management issues of complexity and implicit properties.

Based on the investigation phase, we jointly developed a conceptual view on variability. This conceptual view consists, for example, of:

1. the idea that variation points and dependencies can both be represented as first-class entities,
2. that we can distinguish variation points on different levels of abstraction,
3. that variation points on lower levels realize variation points on higher levels,
4. that dependencies can be modeled on the level of variation points,
5. the relation between impreciseness, incompleteness, and dependencies between variation points.

While we both worked on the investigation phase, and the subsequent development of a conceptual view of variability, Marco was the main responsible for pouring this conceptual view into a modeling language, and a product derivation process.

This research encompassed, amongst others, finding out how a variability language should represent variation points and dependencies, and how it should accommodate for abstraction, incompleteness and impreciseness.

Finally, Sybren was the main responsible for developing the variability assessment technique we mentioned above. This assessment technique was dubbed the COVAMOF Variability Assessment Method (COSVAM), and was designed to address a variety of situations where the question of whether, how and when to evolve variability is applicable. It defines, amongst others, results that can be produced when using this method, defines the way in which information has to be selected and interpreted, and provides the steps to make the variability of a product family explicit.

All results we mentioned above have been published. In the following chapter, we explain how the results are presented in this thesis, and how they are related to the publications.

1.7. References

ACM and IEEE, 1999. Software Engineering Code of Ethics and Professional Practice (Version 5.2).

Booch, 1991. Object-Oriented Analysis and Design with Applications, Benjamin-Cummings Publishing Co., Inc, ISBN ISBN:0-8053-0091-0.

Bosch, J., 2000. Design and use of software architectures: adopting and evolving a product line approach. Pearson Education (Addison-Wesley and ACM Press), ISBN 0-201-67494-7.

Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, H., Pohl, K., 2001. Variability issues in software product lines. In: Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), pp. 11–19.

Brooks, F.P., 1986. No Silver Bullet - Essence and Accident in Software Engineering, Proceedings of the IFIP Tenth World Computing Conference, pp. 1069-1076.

Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. In: SEI Series in Software Engineering. Addison-Wesley. ISBN: 0-201-70332-7.

Clements, P.; McGregor, J.; & Cohen, S., 2005. The Structured Intuitive Model for Product Line Economics (SIMPLE) (CMU/SEI-2005-TR-003, ADA441881). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005.

ConIPF, 2003. Configuration in Industrial Product Families, contract no. IST- 2001-3448. Available from http://search.cs.rug.nl/conipf.

Dixit, A.K., Stiglitz, J.E., 1977. “Monopolistic Competition and Optimum Product Diversity.” American Economic Review, Vol. 67, No. 3, pp. 297-308.

Gibbs, W.W., 1994. Software's Chronic Crisis, Trends in Computing, Scientific American.

IBM, 2001. IBM’s Perspective on the State of Information Technology, available for download at http://www.research.ibm.com/autonomic/index.html, October 2001.

Jazayeri, M., Ran, A., Linden, F. van der, 2000. Software Architecture for Product Families: Principles and Practice. Addison-Wesley.

Leveson N., Turner, C.S., 1993. An Investigation of the Therac-25 Accidents, IEEE Computer, Vol. 26, No. 7, pp. 18-41.

Linden, F. van der, 2002. Software Product Families in Europe: The Esaps & Café Projects. IEEE Software Vol. 19 No. 4, 41–49.

Linden, F. van der, Schmid, K., Rommes, E., 2007. Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering, Springer, ISBN 978-3-540-71436-1.

Lions, J.L., 1996. ARIANE 5 Flight 501 Failure, Report by the Inquiry Board.

Rijksuniversiteit Groningen, http://www.rug.nl.

Robert Bosch Gmbh, http://www.bosch.com.

Robson, C., 2002, Real World Research: A Resource for Social Scientists and Practitioner-researchers, Blackwell Publishing Limited, ISBN 0631213058.

Thales Nederland B.V., http://www.thalesgroup.com/netherlands.

University of Hamburg, http://www.uni-hamburg.de.

Weiss, D.M., Lai, C.T.R., 1999. Software Product-Line Engineering: A Family Based Software Development Process, Addison-Wesley, ISBN 0-201-694387.

Chapter 2 Overview

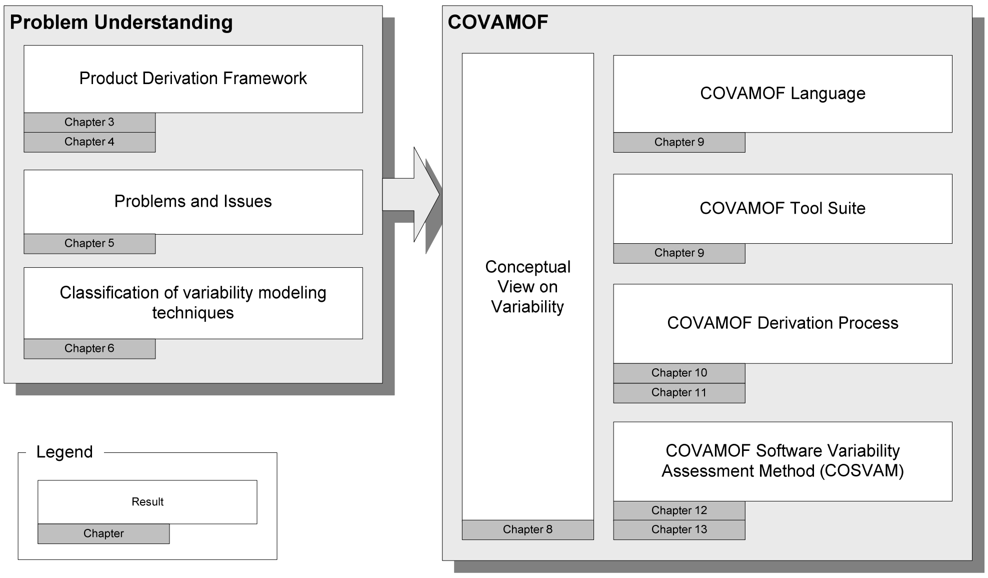


Figure 4. The distribution of results over chapters in this Thesis.

This thesis is an article-based thesis. It is based on one workshop, five conference, and four journal articles. These articles cover our entire research track, viz. from problem identification, to the development of our variability management framework COVAMOF, and its validation. These articles were published as self-contained articles. If we would combine them so that each article would be a chapter, the result would be a thesis that does not read like a smooth story, and which contains duplicate sections. To assist the reader, we therefore chose to combine the articles into chapters, in such a way that the duplicate sections were removed. Each chapter is linked through the abstract. In this chapter, we present the structure of our thesis (see also Figure 4 and Table 3), and show how these articles map to this structure.

Table 3. Link between research questions, results, and chapters.

Research Question	Result	Chapter
Q1 Overview	Product Derivation Framework	Chapter 3 and 4
Q2 Investigation	Problems and issues	Chapter 5
Q3 Variability Modeling Classification	Classification of variability modeling techniques	Chapter 6
Q4 Variability Modeling Technique	Conceptual view on variability	Chapter 7
	COVAMOF Language	Chapter 8
	COVAMOF Tool-suite	Chapter 8
	COVAMOF Derivation Process	Chapter 9 and 10
Q5 Variability Assessment Technique	COVAMOF Software Variability Assessment Method (COSVAM)	Chapter 11 and 12

2.1. Part I – Problem Analysis

Abstract. We started our research by studying product derivation in industrial product families. The goal of this study was to understand the ins- and outs of product derivation, and identify why organizations are involved in a situation where product derivation is too expensive and takes too long.

Part I is the result of this first research phase. It describes a framework of concepts that explains product derivation, a description of case studies in terms of this framework, and the identification of typical product derivation problems that cause expert dependency, high costs, and long time-to-market, and a classification of methods that, supposedly, address these issues.

Chapter 3 – Product derivation framework

Chapter 3 is based on the following workshop and journal articles:

- S. Deelstra, M. Sinnema, J. Bosch, A Product Derivation Framework for Software Product Families, 5th Workshop on Product Family Engineering, Springer Verlag Lecture Notes in Computer Science Vol. 3014 (LNCS 3014), pp. 473-484, May 2004.
- S. Deelstra, M. Sinnema, J. Bosch, Product Derivation in Software Product Families; A Case Study, Journal of Systems and Software, Vol. 74/2 pp. 173-194, January 2005.

As the latter article is an extension of the first article, all sections in these chapters were taken from the latter article.

Chapter 4 – Case studies

This chapter is a combination of case descriptions from the following articles:

- S. Deelstra, M. Sinnema, J. Bosch, Product Derivation in Software Product Families; A Case Study, Journal of Systems and Software, Vol. 74/2 pp. 173-194, January 2005.
- M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, COVAMOF: A Framework for Modeling Variability in Software Product Families, Proceedings of the 3rd Software Product Line Conference (SPLC 2004), Springer Verlag Lecture Notes in Computer Science Vol. 3154 (LNCS 3154), pp. 197-213, August 2004.
- M. Sinnema, S. Deelstra, Industrial Validation of COVAMOF, Elsevier Journal of Systems and Software, Vol 81/4, pp. 584-600, 2007.

The case descriptions of Thales Nederland B.V. and Robert Bosch GmbH were taken from the first article and the Dacolian B.V. case study is a combination of the latter two articles.

Chapter 5 – Problems and Issues

The sections in this chapter were published as Section 5, 6, and 7 of:

- S. Deelstra, M. Sinnema, J. Nijhuis, J. Bosch, Experiences in Software Product Families: Problems and Issues during Product Derivation, Proceedings of the 3rd Software Product Line Conference (SPLC 2004), Springer-Verlag Lecture Notes in Computer Science Vol. 3154 (LNCS 3154), pp. 165-182, August 2004.

As the other sections in this article contained a summary of the product derivation framework and case description of chapter 3 and 4, these sections were left out in this chapter.

Chapter 6 – A classification of variability modeling techniques

All sections in this chapter were published as the following journal article:

- M. Sinnema, S. Deelstra, Classifying Variability Modeling Techniques, Journal on Information and Software Technology, Vol. 49, pp. 717-739, July 2007.

COVAMOF was also part of the original classification in this article. In this thesis, it is described in much more detail in Part II, and was therefore removed from the classification in Chapter 6.

2.2. Part II – Modeling

Abstract. In Part I, we discussed the causes of product derivation issues, and presented a classification that shows the similarities, differences and open issues of variability modeling techniques. These results clearly show that there is enough room for improvement. Based on these results, we therefore developed a collection of techniques called COVAMOF. COVAMOF is a variability management framework that consists of our conceptual understanding of variability, a variability modeling language, a tool-suite, a process, and an assessment technique. This Part presents the modeling concepts and language of COVAMOF. Throughout this Part, we use the Dacolian B.V. case study from Chapter 4 to illustrate our framework.

Chapter 7 – Modeling Concepts

This chapter is a discussion on the relation between dependencies and product derivation from the following conference article:

- M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, Modeling Dependencies in Product Families with COVAMOF, Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2006), March 2006.

The first two sections of this article explain this relation, and were therefore used in this chapter. The discussion of the COVAMOF Meta-model in this article was combined with other articles, and moved to the next chapter.

Chapter 8 – The COVAMOF Language

This chapter is based on the following articles:

- M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, COVAMOF: A Framework for Modeling Variability in Software Product Families, Proceedings of the 3rd Software Product Line Conference (SPLC 2004), Springer Verlag Lecture Notes in Computer Science Vol. 3154 (LNCS 3154), pp. 197-213, August 2004.
- M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, Modeling Dependencies in Product Families with COVAMOF, Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2006), pp. 299-307, March 2006.
- M. Sinnema, S. Deelstra, Industrial Validation of COVAMOF, Elsevier Journal of Systems and Software, Vol 81/4, pp. 584-600, 2007.
- M. Sinnema, S. Deelstra, P. Hoekstra, The COVAMOF Derivation Process, Proceedings of the 9th International Conference on Software Reuse (ICSR

2006), Springer-Verlag Lecture Notes in Computer Science Vol. 4039 (LNCS 4039), pp. 101-114, June 2006.

All these articles describe the modeling concepts of COVAMOF. From the first article, we included the examples, while from the second we included the descriptions of the modeling concepts. The third and fourth article had an introduction and some figures that we included in this chapter.

2.3. Part III – Derivation

Abstract. Based on the concepts and modeling language of Part II, we developed a product derivation process. This process is an instance of the generic product derivation process we discussed in Part I, and is called the COVAMOF Derivation Process. It is specifically tuned to the elements of our framework. In this Part, we present this process and its validation.

Chapter 9 – The COVAMOF Derivation Process

This chapter was published as part of the following conference article:

- M. Sinnema, S. Deelstra, P. Hoekstra, The COVAMOF Derivation Process, Proceedings of the 9th International Conference on Software Reuse (ICSR 2006), Springer-Verlag Lecture Notes in Computer Science Vol. 4039 (LNCS 4039), pp. 101-114, June 2006.

The first three sections of this article contained an introduction, description of a case study, and a discussion of the COVAMOF Meta-model. These parts were already discussed in previous chapters. We also excluded a section on the initial validation of COVAMOF, as it is more extensively discussed in Chapter 10.

Chapter 10 – Validation of COVAMOF during Product Derivation

This chapter was published as part of the following journal article:

- M. Sinnema, S. Deelstra, Industrial Validation of COVAMOF, Elsevier Journal of Systems and Software, Vol 81/4, pp. 584-600, 2007.

As the first sections of this article contain a problem statement that summarize our findings of Part I, and the Dacolian B.V. case description of Chapter 4, only the sections on the hypotheses, experiment set-up, results, and conclusions were included in this chapter.

2.4. Part IV – Evolution

Abstract. The last part of our variability management framework is directed towards evolution of product families. As we noted in the Introduction to this thesis, an alternative approach to decreasing application engineering cost is to make sure there are less mismatches between the variability provided by a product family and the variability required by the products. This Part presents the background, contents, and experiences of applying the COVAMOF Variability Assessment Method (COSVAM). COSVAM is the first technique for assessing variability with respect to the needs of a set of product scenarios. The five steps of COSVAM (identify assessment goal, specify provided variability, specify required variability, evaluate variability, interpret assessment results) form a structured technique that can be tuned to address a variety of situations where the question of whether, how and when to evolve variability is applicable.

Chapter 11 – Variability Assessment and Chapter 12 – COSVAM

This chapter is based on the following conference and journal article:

- S. Deelstra, M. Sinnema, J. Nijhuis, J. Bosch, COSVAM: A Technique for Assessing Software Variability in Software Product Families, Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004), pp. 458-462, September 2004.
- S. Deelstra, M. Sinnema, J. Bosch, Variability Assessment in Software Product Families, Journal of Information and Software Technology, conditionally accepted, 2007.

As the latter is an extension of the first article, we only included the sections of the latter article. We excluded the section that described COVAMOF, because it was already extensively discussed in Part II and Part III of this thesis. We also excluded the discussion on future work because it is presented in the Conclusion of this thesis.

2.5. Conclusion

Abstract. In the conclusion, we reflect on the findings we presented in the previous Parts. We summarize these findings, relate them to our research questions, and discuss their validity. Finally, we discuss the ideas for future work.

PART I. Problem Analysis

We started our Ph. D. training by studying product derivation in industrial product families. The goal of this study was to understand the ins- and outs of product derivation, and identify why organizations are involved in a situation where they experience too much dependency on experts and product derivation is too expensive and takes too long. Part I is the result of this first research phase. It describes a framework of concepts that explains product derivation, a description of case studies in terms of this framework, and the identification of typical product derivation problems that cause expert dependency, high costs, and long time-to-market. In addition, we present a classification of methods that, supposedly, address these issues.

Chapter 3 Product derivation framework

In this chapter, we present a product derivation framework that is meant to define a common language on product derivation. To avoid confusion, we start with a section containing definitions of a number of terms used throughout this Thesis. We continue by presenting a classification for product families, as well as a generic software derivation process. We conclude this chapter by discussing the relation between product family classification and several aspects of product derivation. Combined, these sections build up the product derivation framework that is used in subsequent Parts of this Thesis.

Based on	Section numbers
S. Deelstra, M. Sinnema, J. Bosch, A Product Derivation Framework for Software Product Families, 5th Workshop on Product Family Engineering, Springer-Verlag Lecture Notes on Computer Science Vol. 3014 (LNCS 3014), pp. 473-484, May 2004).	None, superseded by article below
S. Deelstra, M. Sinnema, J. Bosch, Product Derivation in Software Product Families; A Case Study, Journal of Systems and Software, Vol. 74/2 pp. 173-194, January 2005.	All sections in this chapter

3.1. Terminology

We use the following terminology in this document.

Product derivation. A product is said to be derived from a product family if it is developed using reusable product family artifacts. The term product derivation therefore refers to the complete process of constructing a product from product family software artifacts.

Architecture. A product family architecture is the higher level structure that is shared by the product family members. It denotes the ‘‘fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution’’ (IEEE1471, 2000). Each product family member derives its architecture from this overall structure.

Component. A unit of composition with explicitly specified provided, required and configuration interfaces and quality attributes (Bosch, 2000).

Variation point. Places in the design or implementation that identify locations at which variation will occur (Jacobson et al., 1997). Two important aspects related to variation points are binding time and realization mechanism. The term 'binding time' refers to the point in a product's lifecycle at which a particular alternative for a variation point is bound to the system, e.g. pre- or post-deployment. The term 'mechanism' refers to the technique that is used to realize the variation point (from an implementation point of view). Several of these realization techniques have been identified in the recent years, such as aggregation, inheritance, parameterization, conditional compilation (see e.g. Jacobson et al., 1997; Anastasopoulos and Gacek, 2001).

Configuration. A configuration is an arrangement of components and associated options and settings that partially or completely implements a software product. A partial configuration partially implements a software product in the sense that not all variants are selected yet or some variation points are not yet (completely) dealt with. Likewise, a complete configuration is able to fully implement the product requirements, i.e. all necessary variants are selected. In a complete configuration not all variation points have to be bound yet, however. There may still be variation points that are bound at runtime for example.

Knowledge types. We distinct three types of knowledge that are used during product derivation, i.e. tacit, documented, and formalized knowledge. Tacit knowledge (Nonaka and Takeuchi, 1995) is implicit knowledge that only exists in expert minds. Documented knowledge is explicit knowledge that is expressed in informal models and descriptions. Formalized knowledge is explicit knowledge that is written down in a formal language, such as the UML (UML, 2000), and can be used and interpreted by computer applications.

3.2. Product Family Classification

As illustrated in Chapter 1, product families are a successful form of intra-organizational reuse that is based on exploiting common characteristics of related products. In this section, we present a classification of the different types of product families that captures most product families we encountered in practice. This classification consists of two dimensions of scope, i.e. scope of reuse and domain scope.

The first dimension, scope of reuse, denotes to which extent the commonalities between related products are exploited. We identify four levels of scope of reuse, ranging from standardized infrastructure to configurable product base.

- **Standardized infrastructure.** Starting from independent development of each product, the first step to exploit commonalities between products is to reuse the way products are built. Reuse of development methodologies is achieved by standardizing the infrastructure with which the individual applications are built. The infrastructure consists of typical aspects such as the operating system, components such as database management and graphical user interface, as well as other aspects of the development environment, such as the use of specific development tools.

- **Platform.** With a standardized infrastructure in place, the next increase in scope of reuse is when the organization maintains a platform on top of which the products are built. A platform consists of the infrastructure discussed above, as well as artifacts that capture the domain specific functionality that is common to all products. These artifacts are usually constructed during domain engineering. Any other functionality is implemented in product specific artifacts during application engineering. Typically, a platform is treated as if it was an externally bought infrastructure.
- **Software product line.** The next scope of reuse is when not only the functionality common to all products is reusable, but also the functionality that is shared by a sufficiently large subset of product family members. As a consequence, individual products may sacrifice aspects such as resource efficiency or development effort in order to benefit from being part of the product family, or in order to provide benefits to others. Functionality specific to one or a few products is still developed in product specific artifacts. All other functionality is designed and implemented in such a way that it may be used in more than one product. Variation points are added to accommodate the different needs of the various products.
- **Configurable product family.** Finally, the configurable product family is the situation where the organization possesses a collection of shared artifacts that captures almost all common and different characteristics of the product family members, i.e. a configurable asset base. In general, new products are constructed from a subset of those artifacts and require no product specific deviations. Therefore, product derivation is typically automated once this level is reached (i.e. application engineers specify a configuration of the shared assets, which is subsequently transformed into an application).

In addition to the scope of reuse as described above, we identify a second dimension, domain scope. The domain scope denotes the extent of the domain or domains in which the product family is applied.

- **Single product family.** The first domain scope is the individual product family, where a single product family is used to derive several related products.
- **Programme of product families.** In case of a programme of product families, several product families together form a complete system. A shared architecture defines the overall structure of the software systems. The individual components of the system are developed according to an individual

product family approach as described above. The programme of product families is especially applicable for very large software systems.

- **Hierarchical product families.** A hierarchical product family consists of several layers of product families. The top-level product family denotes functionality that is shared by all products, while the lower-level product families specialize the upper levels. This scope is particularly applicable in situations where the number and variability of the involved products is large or very large and a considerable number of staff members is involved in producing those products (Bosch, 2000).
- **Product population.** The product population approach is concerned with reuse of functionality across several domains. Each product family has its own architecture and domain specific components for the required domain specific functionality. Functionality that is shared between the domains is developed in shared components that can be used in the domain specific architectures. For more detailed information on product populations, see Ommering (2002).

3.3. A Generic Product Derivation Process

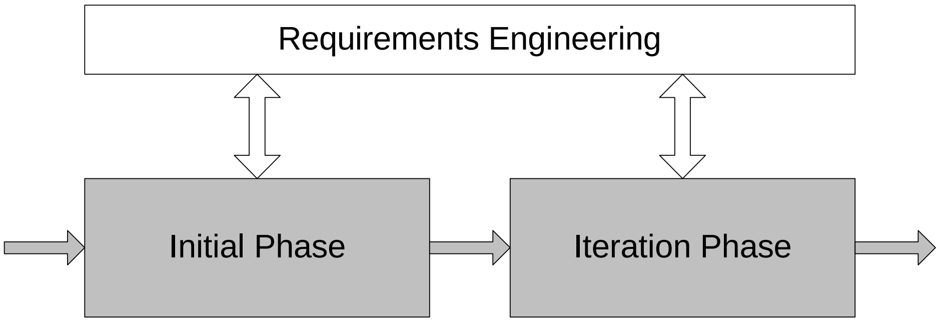


Figure 5. The generic two-phased product derivation process. The shaded boxes denote the two phases of the generic product derivation process. Requirements engineering manages the requirements throughout the entire process.

Focusing on the scope of reuse dimension with a single product family as domain scope, we have generalized the derivation processes we encountered in practice to a generic process as illustrated in Figure 5.

This generic process consists of two phases, i.e. the initial and the iteration phase. In the initial phase, a first configuration is created from the product family assets. During this phase, the application engineer has substantial freedom in choosing alternative product family assets. In the iteration phase, the initial configuration is modified in a number of subsequent iterations until the product sufficiently implements the imposed requirements. The freedom of choice of the application engineer is much more limited during the iteration phase as all decisions have to be made within the context of the product configuration at hand.

In addition to the phased selection activities described above, in all product families, except for product families with the largest scope of reuse, typically some code development is required during product derivation. This adaptation aspect is not strictly bound to one phase in the derivation process. We therefore provide a more detailed description of both phases, as well as a separate description of the adaptation aspect, below.

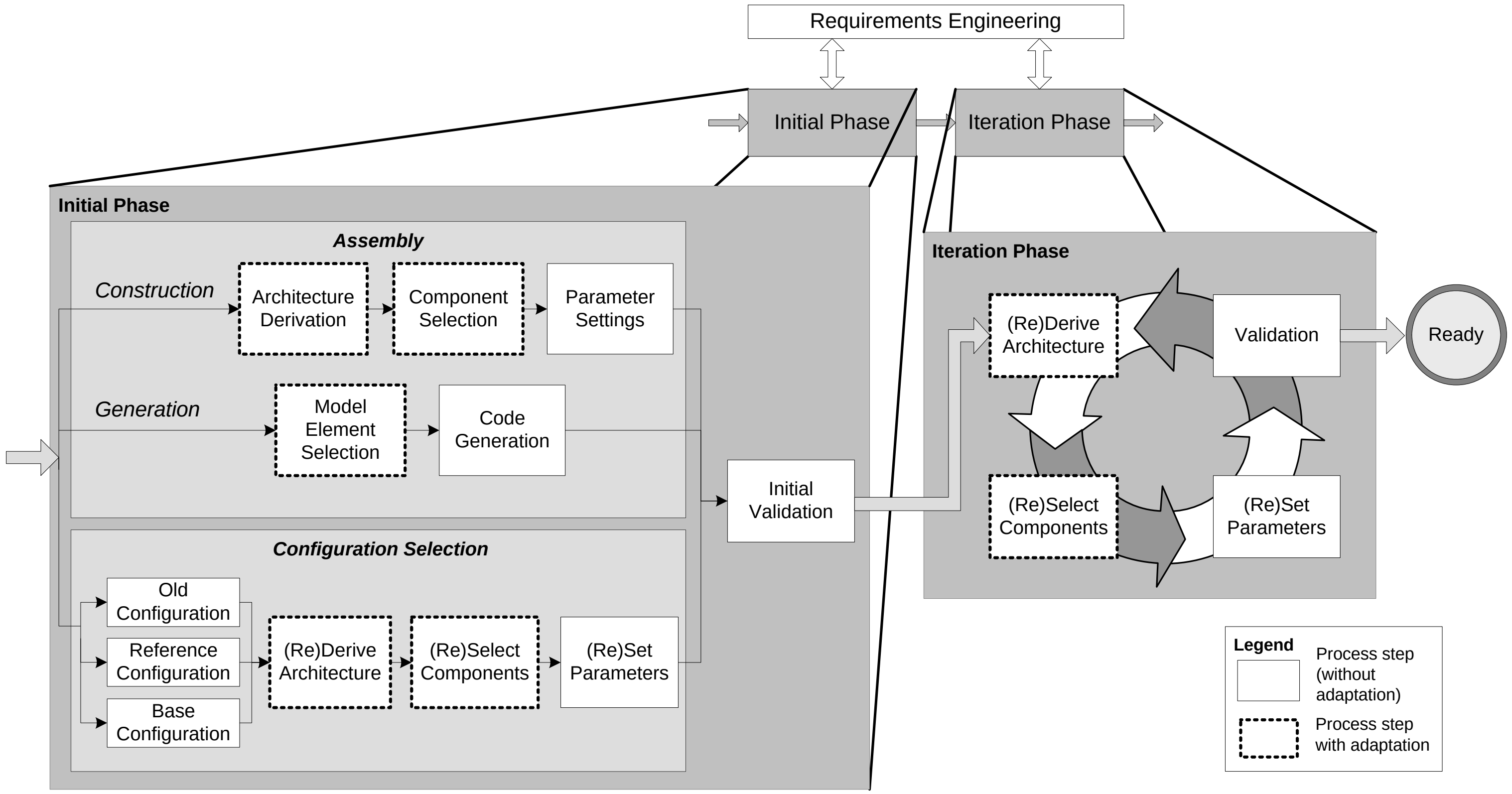


Figure 6. The generic two-phased product derivation process in detail. During the initial phase of the process, a first product configuration is derived from the product family artifacts. Until the product is finished, the initial configuration is modified in a number of subsequent iterations during the iteration phase. Requirements that cannot be accommodated by existing artifacts are handled by product specific adaptation or reactive evolution (denoted by the dashed boxes).

3.3.1. Initial phase

The input to the initial phase is a (sub)set of the requirements that are managed throughout the entire process of product derivation (see Figure 5). These requirements originate from, among others, the customers, legislation, the hardware and the product family organization. In the initial phase, three alternative approaches towards deriving the initial product configuration exist, i.e. assembly, configuration selection, and a hybrid of the former two approaches (see Figure 6 and Figure 7). The alternative approaches conclude with the initial validation step.

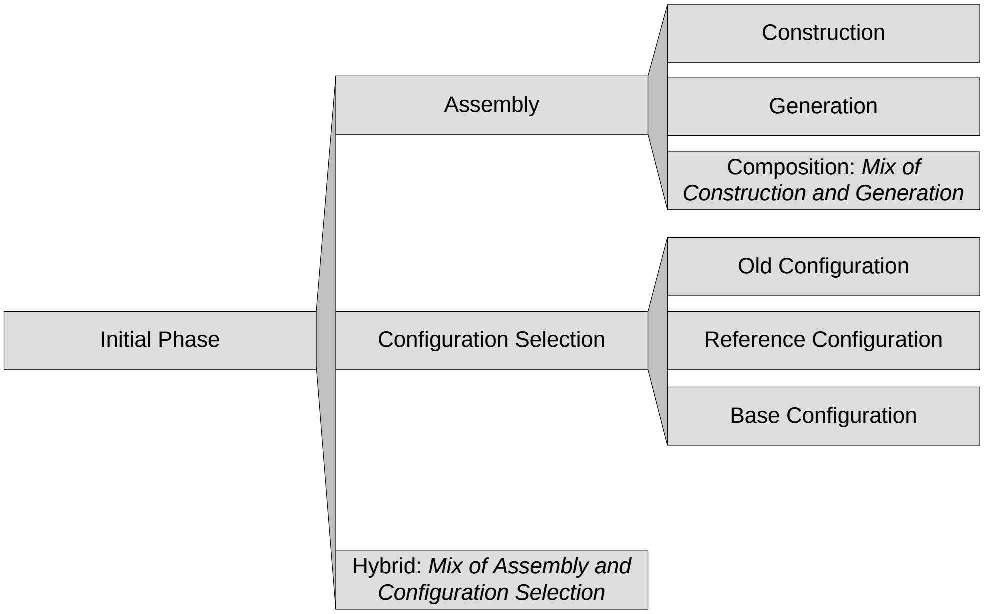


Figure 7. Alternatives during the initial phase. This figure portraits the alternative ways to derive an initial configuration. The composite approach mixes construction and generation, while the hybrid approach mixes assembly and configuration selection.

Assembly. The first approach to initial derivation involves the assembly of a subset of the shared product family assets to the initial software product configuration. We identify three types of assembly approaches.

- In the *construction* approach the initial configuration is constructed from the product family architecture and shared components. The first step in the construction process, as far as necessary or allowed, is to derive the product architecture from the product family architecture. The next step is, for each architectural component, to select the closest matching component implementation from the component base. Finally, the parameters for each component are set.

- In case of *generation*, shared artifacts are modeled in a modeling language rather then implemented in source code. From these modeled artifacts, a subset is selected to construct an overall model. From this overall model an initial implementation is generated.
- The *composition* type (see also Figure 7) is a composite of the types described above, where the initial configuration consists of both generated and implemented components, as well as components that are partially generated from the model and extended with source code.

Configuration selection. The second approach to initial derivation involves selecting a closest matching existing configuration. An existing configuration is a consistent set of components, viz. an arrangement of components that, provided with the right options and settings, are able to function together.

- An *old configuration* is a complete product implementation that is the result from a previous project. Often, the selected old configuration is the product developed during the latest project as it contains the most recent bug-fixes and functionality.
- A *reference configuration* is (a subset of) an old con figuration that is explicitly designated as basis for the development of new products. A reference configuration may be a partial configuration, for example if almost all product specific parameter settings are excluded, or a complete configuration, i.e. the old configuration including all parameter settings.
- A *base configuration* is a partial configuration that forms the core of a certain group of products. A base configuration is not necessarily a result from a previous product. In general, a base configuration is not an executable application as many options and settings on all levels of abstraction (e.g. architecture or component level) are left open. In contrast to a reference and old configuration, where the focus during product derivation is on reselecting components, the focus of product derivation with a base configuration is on adding components to the set of components in the base configuration.

The selected configurations are subsequently modified by rederiving the product architecture, adding, re- and deselecting components and (re)setting parameters. The effectiveness of configuration selection in comparison to assembly is a function of the benefits in terms of effort saved in selection and testing, and the costs in terms of effort required for changing invalidated choices as a result of new requirements. Configuration selection is especially viable in case a large system is developed for repeat customers, i.e. customers who have purchased a similar type of system before. Typically, repeat customers desire new functionality on top of the functionality they ordered for a previous product. In that respect, configuration selection is basically reuse of choices.

Hybrid. Similar to the composite assembly approach, a hybrid approach to configuration exists that mixes assembly and configuration selection. This hybrid approach involves selecting a number of partial configurations (pre-assembled components or subsystems) that are integrated to a larger assembly.

Initial validation. The initial validation step is the first step that is concerned with determining to what extent the initial configuration adheres to the requirements. In the rare case that the initially assembled or selected configuration does not provide a sufficient basis for further development, all choices are invalidated and the process goes back to start all over again. In case the initial configuration sufficiently adheres to the requirements, the product is finished. Otherwise, the product derivation process enters the iteration phase.

3.3.2. Iteration Phase

The initial validation step marks the entrance of the iteration phase (illustrated in Figure 6). In some cases, an initial configuration sufficiently implements the desired product. In most cases, however, one or more cycles through the iteration phase are required, for a number of reasons.

First, the requirements set may change or expand during product derivation, for example, if the organization uses a subset of the collected requirements to derive the initial configuration, or if the customer has new wishes for the product. Second, the configuration may not completely provide the required functionality, or some of the selected components simply do not work together at all. This particularly applies to embedded systems, where the initial configuration is often a first ‘guess’. This is mainly because the exact physics of the controlled mechanics is not always fully known at the start of the project, and because the software performs differently on different hardware, e.g. due to production tolerances, or approximated polynomial relationships. Finally, the product family assets used to derive the configuration may have changed during product derivation, for example, due to bug fixes.

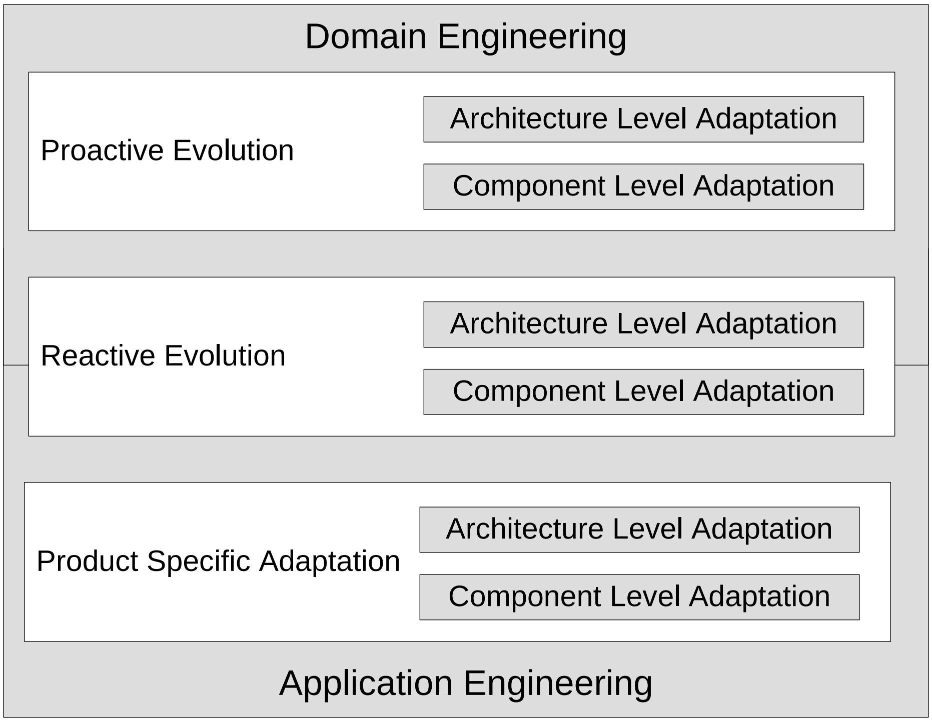
During the iteration phase, the product configuration is therefore modified and validated until the product is deemed ready.

Modification. A configuration can be modified on three levels of abstraction, i.e. architecture, component and parameter level. Modification is accomplished by selecting different architectural component variants, selecting different component implementation variants or changing the parameter settings, respectively.

Validation. The validation step in this phase concerns validating the system with respect to adherence to requirements and checking the consistency and correctness of the component configuration.

Orthogonal to the two phases of the derivation process is the aspect of accommodating requirements that cannot be handled by as-is reuse, i.e. reuse without adaptation, of existing assets. We refer to this activity as adaptation.

3.3.3. Adaptation



Domain Engineering
Proactive Evolution
Architecture Level Adaptation
Component Level Adaptation
Reactive Evolution
Architecture Level Adaptation
Component Level Adaptation
Product Specific Adaptation
Architecture Level Adaptation
Component Level Adaptation
Application Engineering

Figure 8. Three types of artifact evolution. Reactive evolution and product specific adaptation are product derivation activities, while proactive evolution is a 'pure' domain engineering activity.

Although Macala et al. (1996) suggested a five year prediction window for functionality in software product families, in general, products do not precisely adhere to any designed or planned path (Svahnberg and Bosch, 1999). As a result, new product family members may present requirements during product derivation that are not accounted for in the shared product family artifacts. Rather then selecting different artifact variants during the phases described above, these unsupported requirements can only be accommodated by adaptation (denoted by the dashed boxes in Figure 6). Adaptation involves adapting the product (family) architecture and adapting or creating component implementations. We identify three levels of artifact adaptation, i.e. product specific adaptation, reactive evolution and proactive evolution (see Figure 8).

Product specific adaptation. The first level of evolution is where, during product derivation, new functionality is implemented in product specific artifacts (e.g. product architecture and product specific component implementations). To this purpose, application engineers can use the shared artifacts as basis for further development, or develop new artifacts from scratch. As functionality implemented through product specific adaptation is not incorporated in the shared artifacts, it cannot be reused in subsequent products unless an old configuration is selected for those products.

Reactive evolution. Reactive evolution involves adapting shared artifacts in such a way that they are able to handle the requirements that emerge during product derivation, and can also be shared with other product family members. As reactively evolving shared artifacts has consequences with respect to the other family members, those effects have to be analyzed prior to making any changes.

Proactive evolution. The third level, proactive evolution, is actually not a product derivation activity, but a domain engineering activity. It involves adapting the shared artifacts in such a way that the product family is capable of accommodating the needs of the various family members in the future as opposed to evolution as a reaction to requirements that emerge during product derivation. Proactive evolution requires both analysis of the effects with respect to current product family members, as well as analysis of the predicted future of the domain and the product family scope. Domain and scope prediction is accomplished in combination with technology roadmapping (Kostoff and Schaller, 2001).

Independent of the evolution type chosen, the scope of adjustment required on architecture or component level varies in four different ways.

1. **Add variation points.** A new variation point has to be constructed if functionality needs to be implemented as variant or optional behavior, and no suitable variation point is available. In this, we recognize two distinct situations. In the first situation, the new functionality is needed as option or alternative to already implemented stable behavior. This situation mostly occurs if the need for variance was not recognized before or when a change in market conditions forces the organization to support more than one alternative in parallel. In the second situation, the existing system behavior is only extended with the new functionality. In both the situations that the conceptual variant functionality did and did not exist as stable behavior, an interface has to be defined between the variable behavior and the rest of the system. Furthermore, an appropriate mechanism and associated binding time have to be selected and the mechanisms and variant functionality have to be

implemented. In addition, in the situation where existing functionality is involved, the implemented functionality has to be clearly separated from the rest of the system and reimplemented as a variant that adheres to the variation point interface. In case the binding time is in the postdeployment stage, software for managing the variants and binding needs to be constructed.

2. **Change the realization of existing variation points.** Changes to a variation point may be required for a number of reasons. Changes to a variation point interface, for example, may be required to access additional variable behavior. Furthermore, mechanism changes may be required to move the point at which the variant set is closed to a later stage, while a change to the binding time may be required to increase flexibility or decrease resource consumption. In addition, variation point dependencies and constraints may need to be alleviated. In any case, changes to a variation point may affect all existing variants of the variant set in the sense that they have to be changed accordingly in order to be accessible.

3. **Add or change variant.** When the functionality fits within the existing set of variation points, it means that the functionality at a point of variation can be incorporated by adding a variant to the variant set. This can be achieved by extending or changing an existing variant, or developing a new variant from scratch. These new or changed variants have to adhere to the variation point interface, as well as existing dependencies and constraints.

4. **Remove a variant or variation point.** A typical trend in software systems is that functionality specific to some products becomes part of the core functionality of all product family members, e.g. due to market dominance, or that functionality becomes obsolete. The need to support different alternatives, and therefore variation points and variants for this functionality, may disappear. As a response, all but one variant can be removed from the asset base, or the variation point can be removed entirely. If in the latter case one variant is still needed, it has to be re-implemented as stable behavior.

Now that we have established a framework of concepts regarding the derivation process with the two phases and the adaptation aspect, the next question is how these concepts relate to the scope of reuse classification discussed in Section 3.2.

3.4. Relating scope of reuse and product derivation

In the previous section, we stated that products were derived according to a generic process, independent of the scope of reuse. However, there are differences for each of the scopes in the realization of several aspects during product derivation. In this section we discuss those differences.

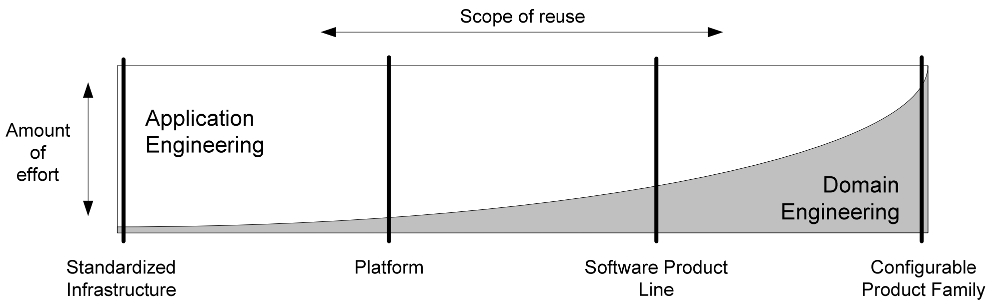


Figure 9. Proportion of cost in four scopes of reuse. This figure presents the division between effort spent in domain and application engineering for the four scopes of reuse discussed in Section 3.2.

3.4.1. Standardized infrastructure

Although it provides a first step towards sharing software assets, a standardized infrastructure provides relatively generic behavior. The infrastructure is typically very stable, and very little domain engineering effort is required. Except for creating and maintaining proprietary glue code, almost all effort is directed towards application engineering (see Figure 9). Therefore, adaptation during product derivation in this type of product family usually only concerns product specific adaptation (see Section 3.3.3).

As the infrastructure contains no domain specific functionality, it cannot fully specify a family architecture, let alone fully document and formalize knowledge to derive a product architecture. It furthermore only documents those parts of component interfaces that are concerned with functionality provided by the infrastructure. Although components may contain variations, variability can be managed as in traditional software development.

3.4.2. Platform

A platform requires a certain amount of domain engineering to create and maintain the assets that implement the common functionality. The main effort, however, is still assigned to application engineering (see Figure 9), and adaptation during product derivation is still mainly concerned with product specific adaptation (see Section 3.3.3).

A platform usually lacks the information about specific products constructed on top of the platform. However, products within a platform based product family require more conformance in terms of architectural rules and constraints that have to be followed. These rules and constraints should be explicitly documented. The component interfaces should also be documented, or at least partially.

Since the platform only captures common functionality, the number of variation points is relatively low. The only variation points available are related to variability that cross-cuts all products. Such variations that are common to all products can be captured and managed explicitly.

3.4.3. Software product line

The amount of effort spent in domain and application engineering is roughly equal in the case of a software product line scope of reuse (see Figure 9). Adaptation during product derivation is concerned with both product specific adaptation and reactive evolution (as described in Section 3.3.3), but the amount of adaptation required highly depends on the stability of the domain and the maturity of the organization.

A software product line provides a product family architecture that specifies both commonalities and differences of the products. For each architectural component, one or more component implementations are provided. For more stable and well understood components, one configurable component implementation exists.

Depending on domain stability and maturity of the organization, for some variation points, the binding time, dependencies and set of variants change frequently, while other variation points are quite stable. Frequent changes make it uneconomical to formalize all knowledge necessary to derive the products. A solution in those situations is to at least partially formalize the specification of the component interfaces for automated consistency checks. The remaining part can remain either documented or tacit.

3.4.4. Configurable product family

A configurable product family typically captures all commonalities and differences in the product base. Most effort in the product family is therefore directed towards proactive evolution. In the occasional event that changes are required they are handled through reactive evolution (see also Section 3.3.3).

The product family architecture is enforced in the sense that no product can or needs to deviate from the commonalities and differences specified in the architecture. The components are often stabilized when this scope of reuse has been reached, and consequently, most components have one configurable component configuration. The return on investment for formalizing knowledge in deriving products using these stable components is therefore substantially higher than in a software product line.

3.5. Related work

After being introduced by Parnas (1976), the notion of software product families has received substantial attention in the research community since the 1990s. The adoption of software product families has resulted in a number books, amongst others, Clements and Northrop (2001), Jacobson et al. (1997), Jazayeri et al. (2000), Weiss and Lai (1999) and Jan Bosch (Bosch, 2000), as well as workshops (PFE 1-4), conferences (SPLC 1 and 2), and several large European projects (ARES, PRAISE, ESAPS and CAFÉ).

Several articles that were published through these channels are related to this chapter. The notion of a variation point was introduced by Jacobson et al. (1997). The notion of variability has also been discussed in earlier work of our research group, amongst others, by presenting three recurring patterns of variability and suggesting a method for managing variability in software product families (van Gurp et al., 2001), discussing variability issues (Bosch et al., 2001), and presenting a taxonomy of variability realization mechanisms (Svahnberg et al., 2002). Several variability realization techniques have further been identified by Jacobson et al. (1997), Jazayeri et al. (2000) and Anastasopoulos and Gacek (2001), while Bachmann and Bass (2001) specifically discuss design and realization of variability in software architectures.

Geyer and Becker (2002) and Salicki and Farcet (2001) present the influence of expressing variability on the application engineering process. They assume a more naive unidirectional process flow, however, rather then a phased process model with iterations for reevaluating the choice for variants. Process models that are more concerned with the development of individual software products have emerged over the years as well. The first known published process model is Royce's waterfall model (Royce, 1970), which is often also used to model the abstraction levels or, alternatively, the lifecycle phases of a software product. Other well known iterative development process models that resemble the ideas of the phased product derivation model we presented in Section 3.3, are the spiral model (Boehm, 1988), the Rational Unified Process (Kruchten, 2000), and the incremental model (Mills et al., 1980).

The two-dimensional maturity classification of product families presented in this chapter is an extension and refinement of the one-dimensional maturity classification presented in earlier work (Bosch, 2002). This one-dimensional classification consisted of the following levels: standardized infrastructure, platform, software product line, configurable product base, programme of product lines, and product populations. The configurable product family relates to the approaches presented by Weiss and Lai (1999) and Czarnecki (1997), who employ generative techniques to derive individual products, and to the Model Driven Architecture approach proposed by the OMG (OMG, 2003). This relationship between product derivation and MDA has furthermore been identified by, for example, Monestel et al. (2002) and Deelstra et al. (2003).

3.6. Conclusion

Software product families can be classified according to two dimensions of scope. The first dimension, scope of reuse, denotes to which extent the commonalities between related products are exploited. The first scope of reuse is the standardized infrastructure, which involves reusing the way products are built. The platform consists of the standardized infrastructure, as well as artifacts that capture the domain specific functionality that is common to all products. In a software product line not only the functionality common to all products is reusable, but also the functionality that is shared by a sufficiently large subset of product family members. As a consequence, individual products may sacrifice aspects such as resource efficiency or development effort in order to benefit from being part of the product family, or in order to provide benefits to others. Finally, the configurable product family is the situation where the organization possesses a collection of shared artifacts that captures almost all common and different characteristics of the product family members, i.e. a configurable asset base.

The second dimension, domain scope, denotes the extent of the domain or domains in which the product family is applied. The first domain scope is the single product family, where a single product family is used to derive several related products. In a programme of product families several product families together form a complete system. A hierarchical product family consists of several layers of product families, and a product population approach is concerned with reuse of functionality across several domains.

Focusing on the scope of reuse dimension in a single product family domain scope, the derivation process that we generalized from practice consists of two main phases, i.e. the initial and the iteration phase (see Figure 6). In the initial phase, a first configuration is created from the product family assets by assembling a subset of shared artifacts or by selecting a closest matching existing configuration. The initial configuration is then validated to determine to what extent the configuration adheres to the requirements imposed by, amongst others, the customer and organization. If the configuration is not deemed finished, the derivation process enters the iteration phase. In the iteration phase, the initial configuration is modified in a number of subsequent iterations until the product sufficiently implements the imposed requirements.

Requirements not accounted for in the shared artifacts are handled by adapting those artifacts. We have identified three levels of adaptation. The first level is product specific adaptation, where new functionality is implemented in product specific artifacts. The second level, reactive evolution, involves reactively adapting shared artifacts in such a way that they are able to handle the new functionality, and can still be shared with other product family members. The third level, proactive evolution, is actually not a product derivation activity, but a domain engineering activity that we added for completeness sake. It involves adapting the shared artifacts in such a way that the product family is capable of accommodating the needs of the various family members in the future.

The terminology, maturity classification, and generic product derivation process we presented in this chapter, are used throughout this thesis. It is a product derivation framework that is meant to serve as a frame of reference to the reader.

3.7. References

Anastasopoulos, M., Gacek, C., 2001. Implementing product line variabilities. In: Symposium on Software Reusability (SSR'01), Toronto, Canada, Software Engineering Notes 26 (3) 109–117.

Bachmann, F., Bass, L., 2001. Managing variability in software architecture. In: Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'01), pp. 126–132.

Boehm, B.W., 1988. A spiral model of software development and enhancement. IEEE Computer 21 (5), 61–72.

Bosch, J., 2000. Design and use of software architectures: adopting and evolving a product line approach. Pearson Education (Addison-Wesley and ACM Press), ISBN 0-201-67494-7.

Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, H., Pohl, K., 2001. Variability issues in software product lines. In: Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), pp. 11–19.

Bosch, J., 2002. Maturity and evolution in software product lines; approaches, artefacts and organization. In: Proceedings of the Second Software Product Line Conference, pp. 257–271.

Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. In: SEI Series in Software Engineering. Addison-Wesley. ISBN: 0-201-70332-7.

Czarnecki, K., 1997. Leveraging reuse through domain-specific architectures. In: Proceedings of the Eighth Workshop on Institutionalizing Software Reuse.

Deelstra, S., Sinnema, M., van Gurp, J., Bosch, J., 2003. Model driven architecture as approach to manage variability in software product families. In: Proceedings of the Workshop on Model Driven Architectures: Foundations and Applications, pp. 109–114.

Geyer, L., Becker, M., 2002. On the influence of variabilities on the application engineering process of a product family. In: Proceedings of the Second Software Product Line Conference, pp. 1–14.

van Gurp, J., Bosch, J., Svahnberg, M., 2001. On the notion of Variability in Software Product Lines, Proceedings of The Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), pp. 45–55.

IEEE, 2000. IEEE Standard P1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems.

Jacobson, I., Griss, M., Jonsson, P., 1997. Software Reuse. Architecture, Process and Organization for Business Success. Addison- Wesley, ISBN: 0-201-92476-5.

Jazayeri, M., Ran, A., Linden, F. van der, 2000. Software Architecture for Product Families: Principles and Practice. Addison-Wesley.

Kostoff, R.N., Schaller, R.R., 2001. Science and Technology Roadmaps. IEEE Transactions on Engineering Management 48 (2), 132–143.

Kruchten, P., 2000. The Rational Unified Process: An Introduction, 2nd ed., ISBN 0-201-707101.

Macala, R., Stuckey, L., Gross, D., 1996. Managing Domain-Specific, Product-Line Development. IEEE Software, 57–67.

Mills, H.D., O'Neill, D., Linger, R.C., Dyer, M., Quinnan, R.E., 1980. The Management of Software Engineering. IBM Systems Journal 19 (4), 414–477.

Monestel, L., Ziadi, T., Jézéquel, J., 2002. Product line engineering: Product derivation, Workshop on Model Driven Architecture and Product Line Engineering, associated to the SPLC2 conference, San Diego.

Nonaka, I., Takeuchi, H., 1995. The Knowledge-Creating Company: How Japanese companies create the dynasties of innovation. Oxford University Press, New York.

OMG, 2000. OMG Unified Modeling Language Specification, Version 1.3, First ed. Available from <http://www.omg.org>.

OMG, 2003. MDA Guide v1.0.

Ommering, R. van, 2002. Building Product Populations with Software Components, Proceedings of the International Conference on Software Engineering 2002, pp. 255–265.

Parnas, D., 1976. On the Design and Development of Program Families. IEEE Transactions on Software Engineering 2 (1), 1–9.

Royce, W.W., 1970. Managing the Development of Large Software Systems: concepts and techniques, Proceedings of the IEEE WESTCON, pp. 1–9.

Salicki, S., Farcet, N., 2001. Expression and Usage of the Variability in the Software Product Lines, Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), pp. 287–297.

Svahnberg, M., Bosch, J., 1999. Evolution in Software Product Lines. Journal of Software Maintenance–Research and Practice 11 (6), 391–422.

Svahnberg, M., van Gurp, J., Bosch, J., 2002. A taxonomy of variability realization techniques. Technical Paper ISSN: 1103-1581, Blekinge Institute of Technology, Sweden.

Weiss, D.M., Lai, C.T.R., 1999. Software Product-Line Engineering: A Family Based Software Development Process, Addison-Wesley. ISBN 0-201-694387.

Chapter 4 Case Studies

As we discussed in the Introduction to this Thesis, we started our research by studying the problems and issues that arise during product derivation in practice. A large part of this study involved the case studies we performed at two industrial partners in the ConIPF project, i.e. Thales Nederland B.V., and Robert Bosch GmbH. Both companies are large and mature industrial organizations that mark two ends of a spectrum of product derivation; Robert Bosch produces thousands of medium-sized products per year, while Thales Nederland produces a small number of very large products. We present these case studies in terms of the product derivation framework we discussed in the previous chapter. In addition to these two case studies, we performed a third case study at Dacolian B.V. This case study is used to exemplify and validate our variability management framework COVAMOF.

Based on	Section numbers
S. Deelstra, M. Sinnema, J. Bosch, Product Derivation in Software Product Families; A Case Study, Journal of Systems and Software, Vol. 74/2 pp. 173-194, January 2005.	Section 4.1 and Section 4.2
M. Sinnema, S. Deelstra, Industrial Validation of COVAMOF, Elsevier Journal of Systems and Software, Vol 81/4, pp. 584-600, 2007.	Section 4.3.1
M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, COVAMOF: A Framework for Modeling Variability in Software Product Families, Proceedings of the 3rd Software Product Line Conference (SPLC 2004), Springer Verlag Lecture Notes in Computer Science Vol. 3154 (LNCS 3154), pp. 197-213, August 2004.	Section 4.3.2

4.1. Thales Nederland B.V.

Thales Nederland B.V., the Netherlands, is a subsidiary of Thales S.A. in France and mainly develops Ground Based and Naval Systems in the defense area. Thales Naval Netherlands (TNNL), the Dutch division of the business group Naval, is organized in four Business Units, i.e. Radars & Sensors, Combat Systems, Integration & Logistic Support, and Operations. Our case study focused on software parts of the TACTICOS naval combat systems family produced by the Business Unit Combat Systems, more specifically, the Combat Management Systems.

4.1.1. The Combat Management Systems product family

A Combat Management System (CMS) is the prime subsystem of a TACTICOS (TACTical Information and COmmand System) Naval Combat System. Its main purpose is to integrate all weapons and sensors on naval vessels that range from fast patrol boats to frigates. The Combat Management System provides Command and Control and Combat Execution capabilities in the real world, as well as training in simulated worlds.

The asset base that is used to build Combat Management Systems, also referred to as the infrastructure, was first established in the 1990s. It provides a virtual machine (referred to as SigMA) as well as a publish/subscribe communication mechanism through a real time distributed database system (referred to as SPLICE). The common functionality is mainly concerned with handling the real time storage and transportation of message instances, the creation of virtual 'worlds' for training, replay of loggings and tests, as well as handling a duplicated LAN and dynamic reallocation of applications for battle damage resistance.

The asset base is now fairly stable, consists of approximately 1500 kLOC and contains both in-house developed and COTS (Commercial Off The Shelf) components. Each component version consists of the source, the binaries, and a fixed documentation associated with it. The descriptions of the components are stored in a hierarchical component repository, where the top-level denotes the functional component and the leaves the small-sized (10–120 kLOC) individual executables.

This component repository and surrounding management tasks play a supportive role within the organization. It mainly provides information and consultancy regarding the use of the components towards several primary processes, handles COTS purchase, component change and maintenance, as well as acceptance of new components to the repository.

Although the Business Unit Combat Systems is in the process of extending the scope of reuse for the Combat Management Systems product family to a software product line, the asset base captures functionality common to all Combat Management Systems and is treated as if it was an externally bought infrastructure. We therefore classify the Combat Management System family as a single product family with a platform as the scope of reuse (see also Section 3.2).

The products built on top of the infrastructure, i.e. the Combat Management Systems, are large and complex technical software systems. An average configuration (including the infrastructure) consists of approximately 37 main components and several MLOC. The infrastructure alone already requires setting several thousand lines of parameters, which often contain multiple parameter settings per line. In the next Subsection, we discuss how the process of deriving the products is organized at TNNL Combat Systems.

4.1.2. Derivation process

The derivation process at the Business Unit Combat Systems is highlighted in Figure 10 and discussed below.

Initial phase. Due to the size of the Combat Management Systems, and the large amount of similarities between configurations for similar types of ships, configuration selection is used to derive the initial product configurations (the thin arrow in Figure 10). To this purpose, the collected requirements are mapped onto an old configuration, whose characteristics best resemble the requirements at hand. In practice, this configuration is usually the most recently completed configuration of the Combat Management System for the same type of ship.

Using this old configuration, a complete specification of the software product is compiled. This specification is used in the rest of the product derivation process. First, the old configuration is modified to comply with the specification. To this purpose, the configuration is modified by re- and de-selecting components, adapting components and changing existing parameter settings.

The architecture is not changed explicitly, but implicitly modified by the selection and adaptation activities. When all components and parameters are selected, adapted and set, the system is packaged and installed in a complete environment for the initial validation. If the configuration does not pass the initial validation, the derivation process enters the iteration phase.

Iteration phase. Until the product sufficiently adheres to the requirements, the configuration is modified in a number of iterations (the thick arrow in Figure 10), by re- and de-selecting components, adapting components and changing existing parameter settings.

Adaptation. During the (re)selection of components, both reactive evolution and product specific changes are applied when components are adapted. The decision whether component development or change will result in product-specific code or development that concerns the entire product family, is determined through Change Control Boards (CCB). Each product development project (which is an application engineering project) has its own CCB that determines whether a request for development for the product family will go to the component CCB (which is part of domain engineering).

The component CCB synchronizes the requests of different projects and decides whether and which of the requests will be honored. The changes are financed and performed by domain engineering. Components are also adapted through proactive evolution. Although all components were well documented at the moment they were initially developed, some of the documentation was not maintained completely when the components were changed. As a result, the derivation process at Combat Systems strongly depends on tacit knowledge. No formal descriptions are used during the derivation process.

The main distinct characteristics of the product families Robert Bosch GmbH are summarized in Table 4. When we relate the description of the product derivation process of TNNL Combat Systems to Section 3.3, evidently, the process at Combat Systems is an instance of the generic derivation process depicted in Figure 6.

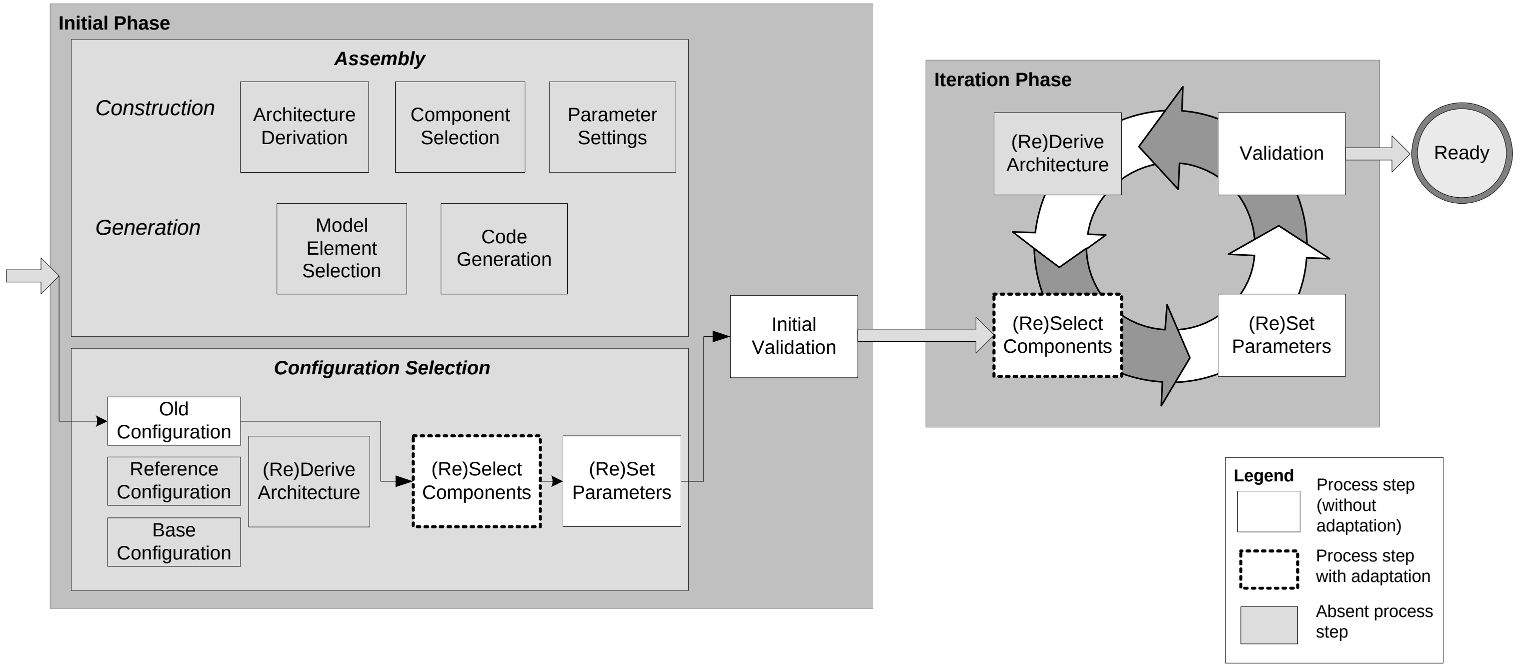


Figure 10. Product derivation at Combat Systems. The customer requirements are mapped onto a closest matching old configuration, which is modified by re- and deselecting components and parameters. After the initial validation, the configuration is modified in a number of iterations, until the product is deemed ready.

4.2. Robert Bosch GmbH

Robert Bosch GmbH, Germany, was founded in 1886. Currently, it is a world-wide operating company that is active in the Automotive, Industrial, Consumer Electronics and Building Technology areas. Our case study focused on two business units, which, for reasons of confidentiality, we refer to as business unit A and B, respectively.

4.2.1. The product families

The systems produced by the business units consist of both hardware, i.e. sensors and actuators, and software. The main requirements for originate from regulations at various parts of the world, and different market segments (e.g. low-cost, mid-range or high-end).

Product family A captures both common and variable functionality of the product family members. Product family A is therefore classified as a family with the software product line as scope of reuse. The asset base of the product family B includes functionality that is common to many products in the family. It has a heartbeat of two months, which means every two months the set of shared artifacts is updated with customer specific functionality that is deemed useful for most other product family members. The scope of reuse for the product family of business unit B is therefore classified as a platform (see also Section 3.2).

While TNNL Combat Systems develops a few instances of the large and complex Combat Management Systems per year, the business units derive thousands of instances of systems per year. In the following Subsections, we discuss the derivation process of both business units.

4.2.2. Derivation process for business unit A

A system that is derived at business unit A, on average contains approximately 50 components, a few hundreds of compiler-switches and several thousands of runtime parameters. The process that is used for deriving these products is highlighted in Figure 11 and discussed below.

Initial phase. Starting from requirements engineering, business unit A uses two approaches in deriving an initial configuration of the product, i.e. one for lead products and one for secondary products:

- **Lead products.** For lead products, the first configuration of the product is constructed using the assembly approach (upper thin arrow in Figure 11). First, the architecture is derived from the product family architecture. In the next step, the closest matching component implementations are selected from the repository and subsequently the parameters are set. If, during the selection of the components, no suitable component implementation can be found in the repository, the most suitable component is copied and adapted where necessary. If the required change involves a completely new concept, the component is developed from scratch (denoted by the dashed box around Component Selection in Figure 11).

- **Secondary products.** In case a similar product has been built before, the first version of the product is developed by using reference configurations (lower thin arrow in Figure 11). These reference configurations are lists of components for consistent configurations, and are typically configurations resulting from previous projects that are explicitly designated as reference point. The components from the reference configuration are replaced with more appropriate ones where necessary. As the reference configurations are lists of components, all parameters are still open and have to be set before the iteration step. Knowledge about valid parameter settings is available however, which restricts the range of variability for the parameter settings.

After the initial configuration is ready, the software product is packaged and installed on a test bench for calibration and validating the product with respect to completeness, correctness and consistency of the configuration. If the configuration passes the validation test, the product is deemed ready. If new customer requirements come in, or when the configuration does not pass validation, the derivation process enters the iteration phase.

Iteration phase. In the iteration phase, the initial configuration is modified in a number of iterations by reselecting components, adapting components, or changing parameters (the thick arrow in Figure 11). At business unit A, up to half of the time spent in deriving a product is consumed by this phase.

Adaptation. If the inconsistencies or new requirements cannot be solved by selecting a different component implementation, a new product family component implementation is developed through reactive evolution. This is achieved by copying and adapting an existing implementation, or developing one from scratch.

The component documentation for individual products often comprise thousands of pages, which are quite up-to-date. Furthermore, formal descriptions are available of the component interfaces. All other knowledge about the artifacts is available as tacit knowledge. The engineers indicate this tacit knowledge is still vital for the product derivation process.

The main distinct characteristics of the product family of business unit A are summarized in Table 4. When we relate the description of the product derivation process of this business unit to Section 3.3, evidently, the process of unit A is an instance of the generic derivation process depicted in Figure 6.

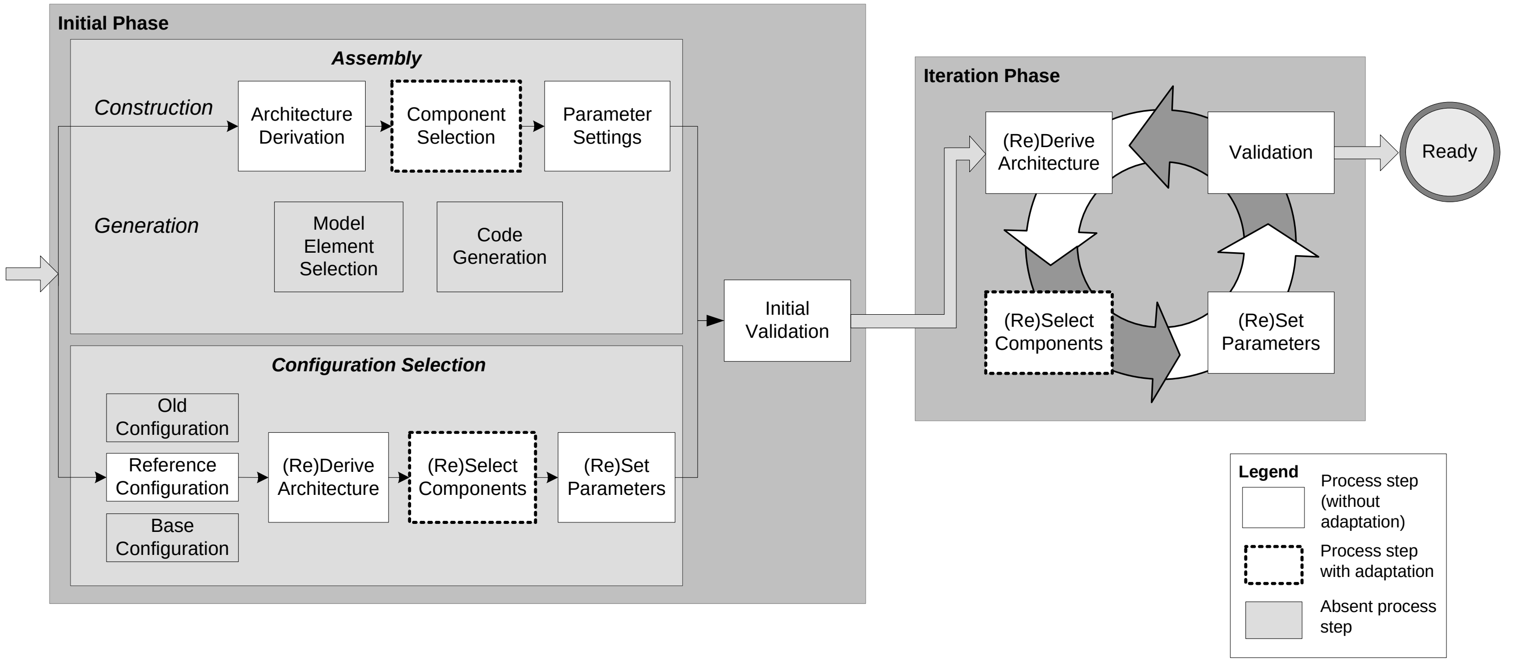


Figure 11 . Product derivation at business unit A. Lead products are derived through assembly, while secondary products are derived using reference configurations. The resulting initial configuration is modified in a number of iterations by reselecting components and resetting parameters. Requirements that cannot be handled using existing components are accommodated by reactively evolving the product family assets during the Component Selection and (Re)Select Components steps.

4.2.3. Derivation process of business unit B

The product derivation process at business unit B is highlighted in Figure 12 and discussed below.

Initial phase. Each time a product needs to be derived from the product family, a project team is formed. This project team derives a product by assembly (the thin arrows in Figure 12). It copies the latest version of the shared components and selects the appropriate components from this copy. Thereafter, all parameters of the components are set to their initial values.

The configuration is then packaged and installed on a test bench for initial validation. If the configuration passes validation, it is deemed ready. Otherwise, the derivation process enters the iteration phase.

Iteration phase. When new requirements come in, or when inconsistencies arise during the validation process, components and parameters are reselected and changed in the iteration phase (the thick arrow in Figure 12), until the product is deemed finished. Similar to the situation at business unit A, the iteration phase at business unit B consumes up to half of the time spent in deriving a product.

Adaptation. When during the initial and the iteration phase the requirements for a product configuration cannot be handled by the existing product family assets, the changes are applied to the selected component copies, in other words, by product specific adaptation. These changes can therefore not be reused unless the new functionality is integrated into the shared artifacts at the heartbeat.

The main distinct characteristics of the product family of business uni B are summarized in Table 4. When we relate the description of the product derivation process above to Section 3.3, evidently, the process at business unit B is an instance of the generic derivation process depicted in Figure 6.

Table 4. Case study characteristics. The main characteristics of the product families at Combat Systems, Thales Nederland B.V., and two business units at Robert Bosch GmbH.

Product family	Scope of reuse	Initial derivation phase	Adaptation
TNNL Combat Systems	Platform	Old configuration	Reactive and product specific
Bosch A	Product line	Reference configuration and assembly	Reactive
Bosch B	Platform	Assembly	Product specific

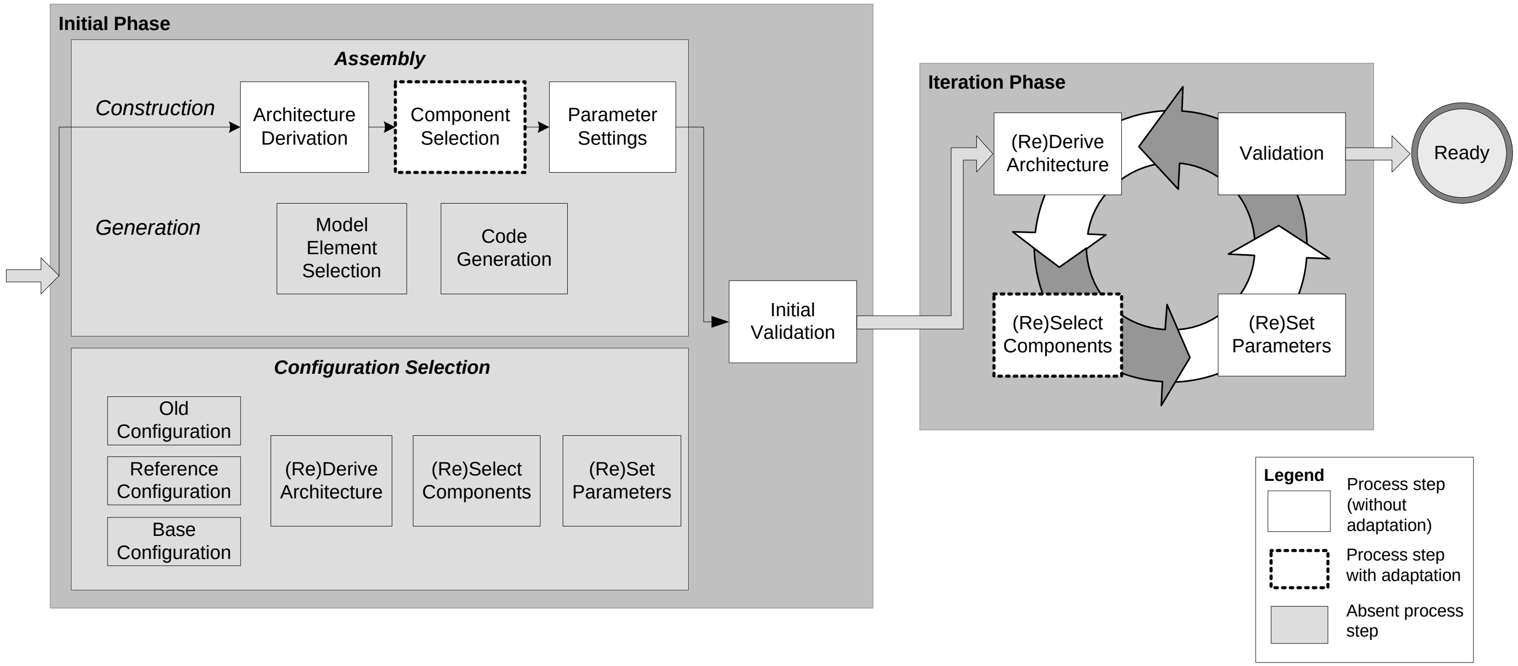


Figure 12. Product derivation at business unit B. Products are derived by assembling an initial configuration. After the initial validation, the configuration is modified in a number of cycles in the iteration phase. Requirements that cannot be handled by existing product family assets are handled through product specific adaptation during the Components Selection and (Re)Select Components steps.

4.3. Dacolian Case Study

In addition to the case study that we used to identify problems and issues, we performed a case study at Dacolian B.V. to validate the variability management framework that we discuss in later parts of this thesis. Dacolian B.V. (see Dacolian Website) is a Dutch company that develops intellectual property software modules for Intelligent Traffic Systems (ITS). In this section, we describe the organization and its product family. Throughout this Thesis, we use examples from this case study to illustrate and validate our COVAMOF variability management framework.

4.3.1. Organization and Product Family

Dacolian B.V. uses a product family called Intrada® to produce the software modules for Intelligent Traffic Systems (ITS). The product family artifacts of this family consist of approximately 11 million lines of code. Typical products consist of systems that deliver; based on real-time input images, abstract information on the actual contents of the images, e.g. the presence of traffic, vehicle type and brand, car registration numbers, etc. As such, Dacolian is worlds leading supplier of OEM software modules for intelligent traffic systems that utilize Automatic License Plate Recognition (ALPR), see Figure 13.

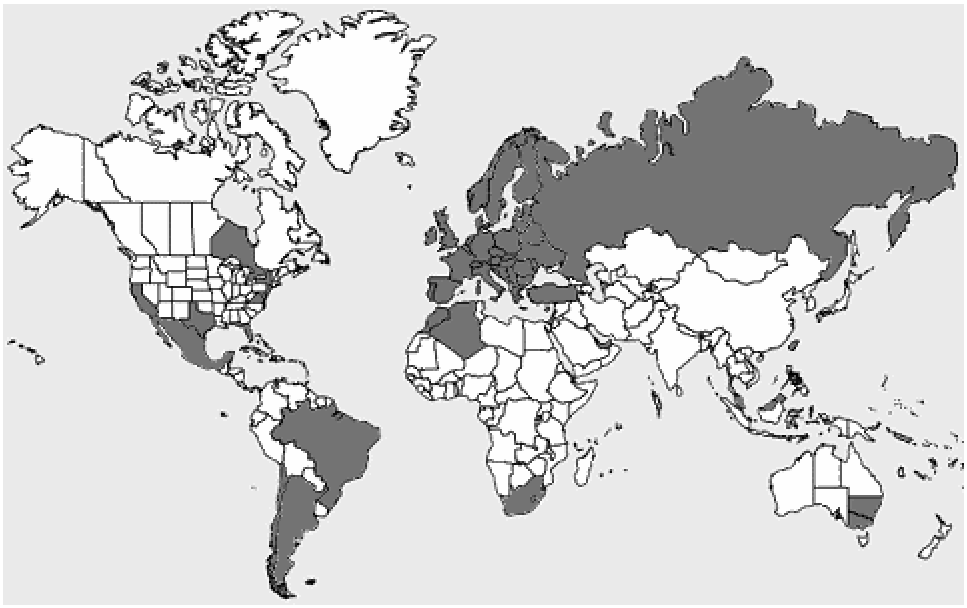


Figure 13. Dacolian Worldwide. This figure depicts an overview of the countries where Dacolian software is used for tolling, law-enforcement, and access control/parking.

Intrada modules can be found within the three major ALPR application areas, which are tolling, enforcement, and access control/parking (see also Figure 14). Examples of (open road) tolling systems around the world are the Highway 407ETR in Canada, the Costanera Norte in Chile, Nationwide Electronic Toll Collection in Taiwan, and several highways in the USA. Examples of the enforcement and parking application areas are the internationally well known Gatsometer cameras that are used for red light and speed enforcement (Intrada modules are embedded within these systems) and the parking systems at the Frankfurt International airport system of VMT Düssel, respectively. Each of these application areas provides its own distinct set of performance requirements and image characteristics resulting in a different configuration of Intrada OEM modules.

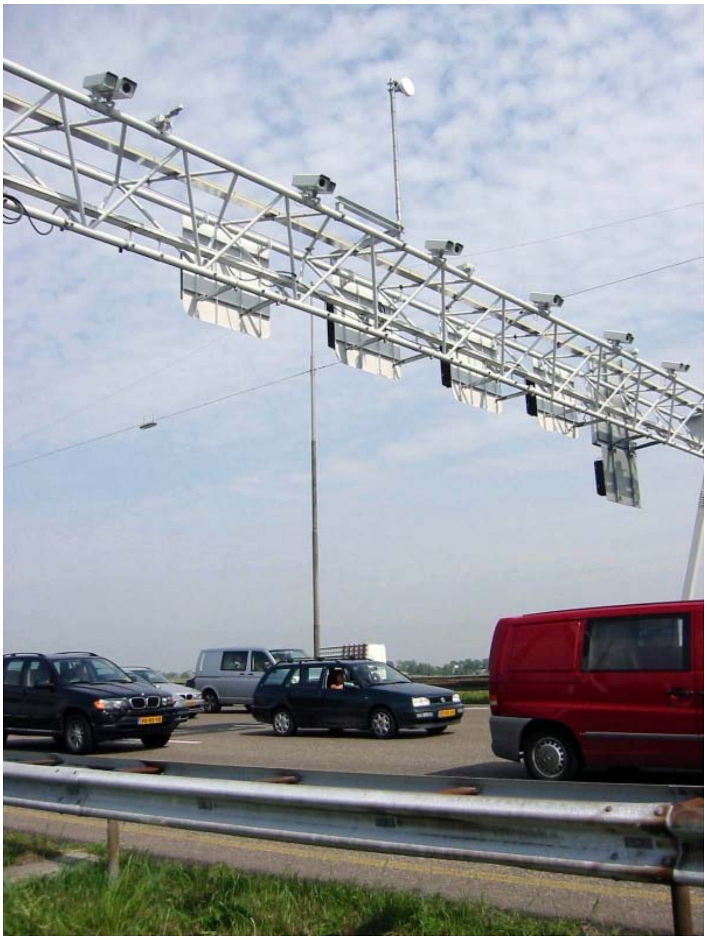


Figure 14. Automatic License Plate Recognition. One of the application areas of automatic license plate recognition of Dacolian B.V. is tolling on highways.

4.3.2. Intrada Product Family Architecture

Dacolian maintains a hierarchical product-family architecture, of which we present the logical view in Figure 15. Dacolian furthermore maintains in-house developed reusable components that are used for product construction. Many of the components are frameworks. Examples of these frameworks are a common interface to a number of real and virtual frame grabbers (Image Intake component in Figure 15) and a framework providing license specialization and enforcement (License Manager component in Figure 15). 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.

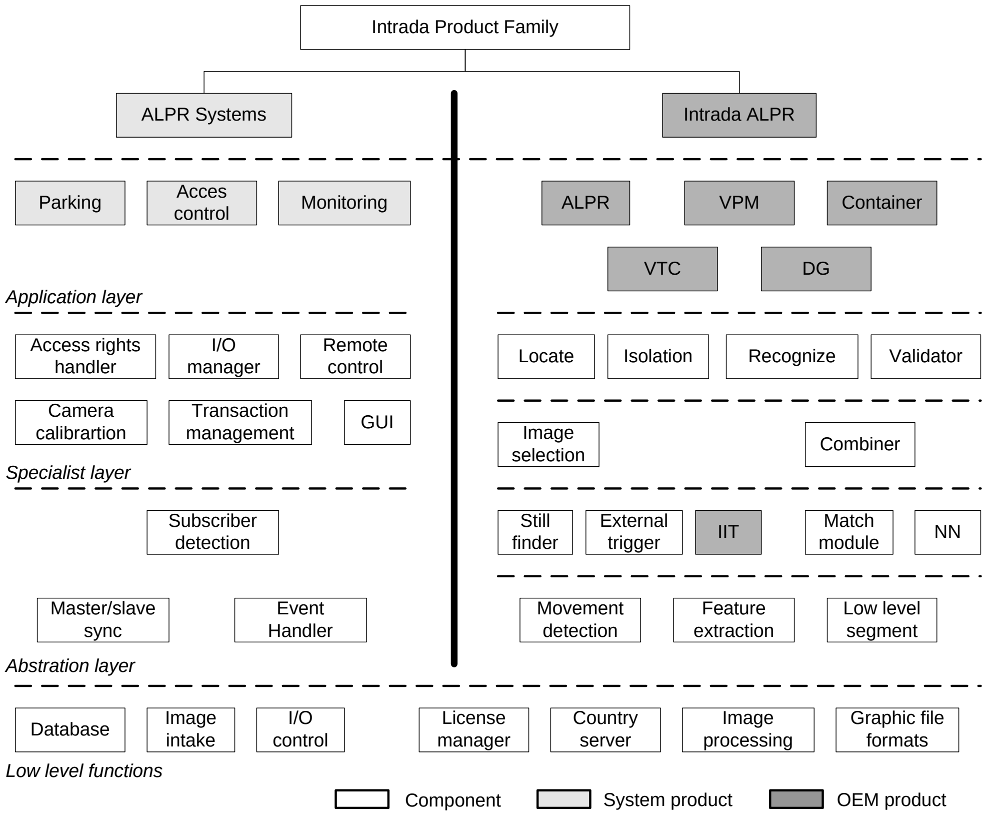


Figure 15. Logical view on the Hierarchical Intrada Product Family.

The responsibilities of the reusable components in Figure 15 are denoted by the layers in the logical view. Each component incorporates variability and uses a selection of the components in a lower abstraction level. The thick solid vertical line represents the separation between the two branches within the Intrada Product Family, i.e. the *Intrada Systems* and *Intrada ALPR* families.

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-products (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 done 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, pre-compiler directives and Make dependencies. Binding at link time is almost not applied in the Intrada product family, and Make dependencies only specify variation at compile time.

4.3.3. Evolution

The Intrada ALPR product family evolved over the past seven years. In Figure 16, we depict the products that were sold during the two years prior to the case study. In that period, the Intrada ALPR product family contained four different products groups (the space dimension), and all product groups evolved over several generations (the time dimension). At the time of the case study, the most recent version of the Intrada ALPR family was 4.0.0, but several customers still had products in the field that had been built with older releases of the Intrada ALPR family.

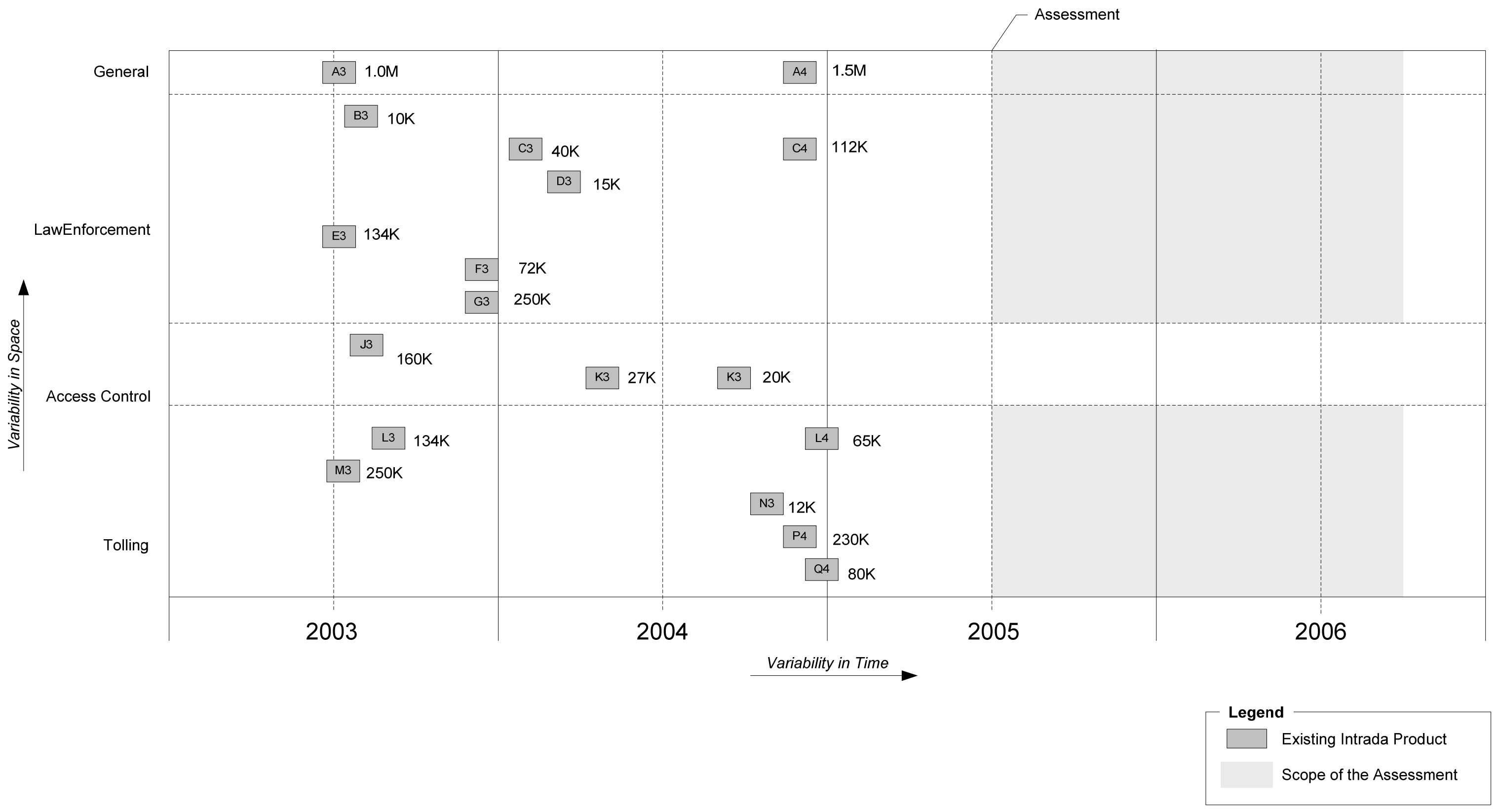


Figure 16. Intrada ALPR product evolution. This figure depicts different products of the Intrada ALPR product family. The graph shows that, the Intrada ALPR product family evolved in both space (increasing number of different products), as well as time (new versions of products).

Chapter 5 Problems and Issues

The main goals of the Bosch and Thales case studies we presented in the previous chapter were to gain an understanding of product derivation processes at large organizations that employ software product families, and to determine the underlying problems and causes of challenges faced during this process. This chapter discusses the problems and issues we identified. We present descriptions, examples, consequences, solutions and research issues. The discussions are relevant outside de context of the two case study organizations, as the challenges they face do or eventually will arise in, for example, comparable or less mature organizations.

Based on	**Section numbers**
S. Deelstra, M. Sinnema, J. Nijhuis, J. Bosch, Experiences in Software Product Families: Problems and Issues during Product Derivation, Proceedings of the Third Software Product Line Conference (SPLC 2004), Springer Verlag Lecture Notes on Computer Science Vol. 3154 (LNCS 3154), pp. 165-182, August 2004.	All sections in this chapter

5.1. Introduction

The case studies at Bosch and Thales consisted of a number of interview sessions with key personnel involved in product derivation, amongst others, system architects, software engineers, and requirement engineers. Although questionnaires guided the interviews, there was enough room for open, unstructured questions and discussions. We recorded the interviews for further analysis afterwards, and used documentation provided by the companies to complement the interviews.

The outline of our discussion is illustrated in Figure 17. We start the discussion with the high-level challenges that served as motivation for investigating product derivation problems. In the observed problems section, we discuss the main problems that were identified during the case study and that underpin those challenges. The core issues are the common aspects of these problems. On the one hand, product derivation problems can be alleviated by process guidance and methodological support that finds a suitable way to deal with these issues. On the other hand, product derivation problems can be alleviated by addressing the underlying causes of the core issues. The core issues and related methodological support issues are discussed in Section 5.4. The underlying causes of the core issues are addressed in the last two Sections (5.5 and 5.6).

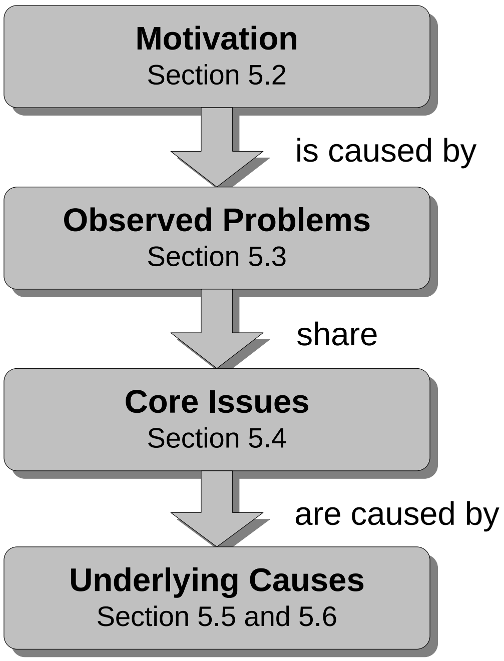


Figure 17. The outline of the discussion in this chapter. The problems we directly observed during our case studies share a set of common aspects. We call these common aspects core issues. Each core issue has a number of underlying causes.

5.2. Motivation

Two main observations prompted the need to investigate product derivation.

Time, effort and costs: The first observation involved the classic software engineering problems, such as high costs associated with deriving individual products, as well as the time and effort consumption because of difficulties in individual product derivation steps and the large number of cycles through the iteration phase.

Expert Dependency: The second observation involved the fact that the product derivation process depends very much on expert involvement. This not only resulted in a very high workload on experts, and as such in the lack of accessibility of these experts, but also made the organizations very vulnerable to the loss of important derivation knowledge.

5.3. Observed Problems

During the data collection phase of the case study, we have identified several underlying problems for the observations discussed in the previous section. In the remainder of this section, we provide a description for each underlying problem, followed by an example.

5.3.1. False positives of compatibility check during component selection

Description: During product derivation, components are selected for addition to or replacement of components in the (partial) configuration at hand. Whether a component fits in the configuration depends on whether the component correctly interacts with the other components in the configuration and whether no dependencies or constraints are violated. The interaction of a component with its environment is contractually specified through the provided and required component interface. Thus, the consistency of the cooperation with the other components can be validated by checking the correctness of the use of the provided interfaces, the satisfaction of the required interfaces, the constraints, and the dependencies of the selected component. The problem we identified is the situation in which the (manual or automated) consistency check does not find any violations, while during testing it turns out some of the selected components are not compatible. As a result, extra iterations and expert involvement are necessary in order to correct the inconsistent configuration.

Example: Part of the compatibility check between components during component selection is performed by tooling at business unit A. This automated check is based on the syntactical interface specification, i.e. function calls with their parameter types. One of the causes for iterations is that although the interface check indicates a positive match for components in the configuration during earlier phases, in some cases, components turn out to be incompatible during validation on the test bench.

5.3.2. Large number of human errors

Description: Due to the large amount of variation points, i.e. ranging up to tens of thousands, and the possibly even larger number of (implicit) dependencies between these variation points, product derivation at both organizations has become an error-prone task. As a result, a substantial amount of effort in the derivation process of the interviewed business units is associated with correcting these human errors in the iteration phase.

Example: Parameter settings at TNNL Combat Systems account for approximately 50 percent of product derivation costs. The engineers indicated that most of the effort for this activity results from correcting unintended side-effects introduced by previous parameter setting steps.

5.3.3. Consequences of variant selection unclear

Description: Software engineers often do not know all consequences of the choices they make in the derivation process. As a result, selected variants may turn out to complicate development.

Example: In addition to automated checks, the interviewed business units check component compatibility and appropriateness by manually inspecting the component documentation. The software engineers indicated that despite the fact that the component documentation for individual products comprises up to thousands of pages, during testing, selected components often insufficiently implement requirements or turn out to be incompatible.

5.3.4. Repetition of development

Description: Despite their reuse efforts, the organizations identified that development effort is spent on implementing functionality that highly resembles functionality already implemented in reusable assets or in previous projects.

Example: The asset repository of business unit A contains several thousand component-variants and several tens of thousands older versions of these component variants. Product derivation at this business unit comprises selecting several hundreds of components from these component-variants and setting several thousand parameters. Occasionally, this leads to a situation where during component selection, the component variants implementing certain requirements are not found, while they are present.

5.3.5. Different provided and required interfaces complicate component selection

Description: Variants of variation points communicate with other parts of the system through the provided and required interfaces. The problem we address here involves the situation where several variants of a particular variation point have different provided and required interfaces. This complicates component selection as in some cases multiple versions of the same variant provide the same required functionality, but differ in provided or required interface, or two variants of different variation points can not be selected both, as they provide incompatible interfaces to each other. As a result, selecting a required variant can invalidate the selection of earlier selected variants or require adaptation.

Example: This problem was primarily identified at business unit A. Complications in component selection occur frequently as a result of different provided and required interfaces, while the components implement the functionality that is required. One of the simpler examples of such a difference is where component A provides a value that specifies a positive delta, while component B expects a negative delta.

5.4. Core Issues

Upon closer inspection, the problems listed above share the combination of two core issues as a common aspect.

Complexity. The first core issue identified is the complexity of the product family in terms of number of variation points and variants. This complexity causes the process of setting and selecting variants to become almost unmanageable by individuals.

Implicit properties. The second core issue involves the large number of implicit properties (e.g. dependencies) of variation points and variants, i.e. properties that are undocumented and either unknown or only known by experts.

5.4.1. Coping with complexity and implicit properties

In themselves, complexity and implicit properties are not problematic. It is due to the failure to effectively deal with them that the issues result in problems discussed in the previous section. On the one hand, product derivation problems can therefore be alleviated by process guidance and methodological support that provides a suitable way to deal with these issues, for example:

Hierarchical organization. One of the reasons why the large number of variation points and variants is a problem, is the fact that none of the product families of the case study explicitly organized variation points in a hierarchical fashion (as for example employed in feature diagrams (Svahnberg et al. 2002)). As a consequence, during product derivation, software engineers are required to deal with many variation points that are either not at all relevant for the product that is currently being derived, or that refer to the selection of the same higher level variant.

First-class and formal representation of variation points and dependencies. The lack of first-class representation of variation points and dependencies makes it hard to assess the impact of selections during product derivation, and changes during evolution, as it is unclear how variations in requirements are realized in the implementation (Bosch et al. 2001). If formalized, first-class representations of variation points and dependencies enable tool support that may significantly reduce costs and increase efficiency of product derivation and evolution.

Reactive documentation of properties. The problems as a result of implicit dependencies are not only associated with the one time occurrence, but also with the fact that mistakes keep reoccurring in several projects. This is due to the lack of documenting implicit properties that frequently result in mistakes. One approach to dealing with implicit properties is therefore through a reactive documentation process that ensures documentation of important implicit dependencies.

On the other hand, rather than focusing on dealing with the issues, product derivation problems can be alleviated by addressing the source of the core issues. During cause analysis, we identified that both core issues have a number of underlying causes. We discuss the underlying causes of complexity and implicit properties in the following two sections, respectively. For each of these causes, we provide a description and consequences, as well as solutions and topics that need to be addressed by further research.

5.5. Underlying causes of complexity

Complexity in product families is both due to the inherent complexity of the problem domain (e.g. complex system properties, number of product family members), as well as the design decisions regarding variability that are made during the evolution of the shared software assets. In the following cause analysis, we focus on increases in complexity that are the result of evolution of variability. We identified issues related to scoping during product derivation, obsolete variation points, and non-optimal realization of variability as underlying causes.

5.5.1. Scoping during product derivation: product specific adaptation versus reactive evolution

Description: As we discussed in our product derivation framework, changes that are required during product derivation (e.g. new features), can be handled through product specific adaptation or reactive evolution. An important aspect related to these types of evolution is scoping, i.e. determining in which of both types a change should be handled.

Such a scoping decision is optimal if the benefits of handling the change through reactive evolution (possible reuse) outweigh the benefits of product specific adaptation (no unnecessary variation points and variants). At Bosch, we identified two extremes in the balance between product specific adaptation and reactive evolution. On the one extreme, all required changes for the products were handled through product specific adaptation (business unit B), while on the other extreme, all changes were handled through reactive evolution (business unit A). In other words, no explicit asset scoping activity existed that continuously interacted with product derivation.

Consequences: The drawback of not individually scoping each feature during product derivation, is that, for possibly a considerable amount of functionality, a non-optimal scoping decision is made, viz. a decision either resulting in repetition of development in different projects, or in more variation points and variants than actually needed. The lack of scoping during product derivation is thus the first underlying cause of complexity.

Solutions and research issues: Product derivation processes should include continuous interactions with the scoping activities during domain engineering. In that respect, the way Change Control Boards are employed in TNNL Combat Systems (see case descriptions) provides a good example for large product families. For smaller product families, this continuous interaction is the responsibility of the product family architect(s). Currently, however, there is a lack of variability assessment techniques that consider both the benefits and the drawbacks of scoping decisions. We consider these techniques to be crucial in assisting the system engineers and architects during evolution.

5.5.2. Proactive evolution: obsolete variation points

Description: The problem we address here is twofold. First, a typical trend in software systems is that functionality specific to some products becomes part of the core functionality of all product family members, e.g. due to market dominance. The need to support different alternatives, and therefore variation points and variants for this functionality, may disappear. Second, variation points introduced during proactive evolution are often not evaluated with respect to actual use.

Both phenomena lead to the existence of obsolete variation points, i.e. variation points for which in each derived product the same variant is chosen, or, in case of optional variants, not used anymore. At both organizations, these obsolete variation points were often left intact rather than being removed from the assets.

Consequences: Removing obsolete variation points increases the predictability of the behavior of the software (Bosch et al. 2001), decreases the cognitive complexity of the software assets, and improves traceability of suitable variants in the asset repository. We also note that if variability provided by artifacts is not used for a long time and removed from the documentation, engineers may start to forget some facilities are there. For example, the interviewees indicate the existence of parameters from which no one knows what they are for, let alone what the optimal value is. In addition to handling all changes through reactive evolution, we therefore identify the existence of obsolete variation points and the lack of removing these obsolete variation points, as the second underlying cause of complexity.

Solutions and research issues: The solution to this issue is not as simple as just removing the variation point. Besides the fact that it may prove to be hard to pinpoint the exact moment at which a variation point becomes obsolete ("but it might be used again next month!"), removing obsolete variation points requires effort in many areas. All variation points related to the previously variable functionality have to be removed, which may require redesign and reimplementation of the variable functionality. There may be dependencies and constraints that have to be taken care of. Also, changing a component may invalidate existing tests and thus require retesting. In order to reduce complexity, we identify the need for methodologies regarding road mapping and variability assessment that do not only consider the necessity and feasibility of including new functionality in the shared software assets, but that also regularly consider the removal of unused functionality (and thus variation points and variants).

5.5.3. Non-optimal realization of variability

Description: Over the past few years, several variability realization techniques have been identified (Jacobson et al. 1997; Svahnberg et al. 2002). Variability realization mechanisms typically have a large impact on the performance and flexibility of a software system. Therefore, a well-considered choice should be made when selecting a mechanism, based on aspects such as the size of the involved software entities, when the variation should be introduced, when it should be possible to add new variants, and when it needs to be bound to a particular variant. In practice, however, the number of different variability realization techniques used is often very limited and not always best suited for the problem at hand. We identify a number of reasons:

- As some variation mechanisms require extra computational resources such as memory or CPU-cycles, especially organizations that produce embedded systems often choose mechanisms that minimize the overhead.
- The number of variability mechanism used is further limited by the technology used. The programming language C for example, does not allow for techniques such as inheritance.
- Software architects typically lack the awareness with respect to the advantages and disadvantages of certain mechanisms and often only map variation to the mechanisms they know (Bosch et al. 2001).
- Customer requirements may require a certain mechanism to be used, whereas this mechanism is not recommended from a technology point of view. An example of such a situation is when runtime binding is prohibited, as the code is restricted to be shipped to one customer only.
- Reactive evolution is mainly concerned with incorporating the requirements that are specific to a product in relation to existing product family requirements, rather than analyzing how they relate to requirements in future products. In addition, reactive evolution suffers from time-to-market and

budget pressure of individual projects. With this focus, implementing variation points and variants for a specific product is prioritized over reusability in other products.

Consequences: Although choosing only a few mechanisms simplifies design and implementation, the drawbacks of disregarding the other mechanisms are not always understood (Bosch et al. 2001). In the interviewed business units, most variation is mapped to variant component implementations and parameterization. The drawback of this approach for business unit A, for example, is that even small changes to the interfaces needed to be implemented through new component implementations. This contributed to the fact that the asset repository of business unit A now contains several thousand component-variants and several ten-thousand older versions of these component variants. Realization of variability is therefore identified as a third underlying cause of complexity.

Solutions and research issues: We identify the need for detailed comparative studies of realization techniques in order to understand the qualities and drawbacks of selecting certain mechanisms. Adequate training of software architects with respect to design for variability helps to create awareness of different variability mechanisms.

5.6. Underlying causes of implicit properties

The second core issue, implicit properties, is related to the creation and maintenance of documentation. In the following cause analysis, we identify three underlying causes, i.e. causes related to erosion, to insufficient and over-explicit documentation, and to the lack of specifying semantics and behavior.

5.6.1. Erosion of documentation

Description: Documentation has to be actively maintained in order to keep it useful. At the original development of software artifacts, the specification can be correct, or at least intended to be correct. If in a later stage an artifact is changed, the specification should also be changed accordingly. Quite often however, investigating all consequences of changes and updating the documentation accordingly is not a priority.

Consequences: The severity of situations where documentation shows signs of erosion, range from minor inconsistencies that result in errors, and thus iterations, up to the stage were it is out-dated and rendered almost useless. Eroded documentation furthermore contributes to, for example, the fact that it requires several years to train personnel at TNNL Combat Systems.

Solution and research topics: The issue discussed here, confirms the documentation problem regarding reuse discussed by Bosch (1999). Solutions and research issues suggested in that article, such as not allowing engineers to proceed without updating documentation, a higher status and larger amount of support from management, and investigating novel approaches to documentation, therefore also apply here. A technique that specifically addressed erosion is literate programming (Knuth 1984), where source code and documentation live in one document. Literate programming is state of practice through, for example, Javadoc (2003), and may provide a large and easy step forward for many C oriented systems.

5.6.2. Insufficient, irrelevant and voluminous documentation

Description: Software engineers often find it hard to determine what knowledge is relevant and should be documented. Although problems with documentation are usually associated with the lack of it, the large amount and irrelevance of documentation was identified as a problem as well.

Consequences: In addition to the automated check, component compatibility at the interviewed business units is performed manually by inspecting the component documentation. The software engineers indicated that the component documentation for individual products often comprise thousands of pages. On the one hand, the volume and irrelevancy of the information contained in the documents made it hard to find aspects (e.g. dependencies) relevant for a particular configuration. On the other hand, the software engineers also indicate that, at times, even the large amounts of relevant documentation seem not enough to determine the provided functionality or compatibility.

Solution and research topics: Besides good structuring, documentation requires an acceptable quality to quantity ratio in order to be effective. Relevant research issues have been discussed in the research issues for erosion of documentation.

5.6.3. Lack of semantics and behavior

Description: Component interface specifications often consist of a set of provided and required operations in terms of argument and return types. Such specifications do not specify the semantics and behavior of the interface.

Consequences: One of the reasons for the false positive problem as discussed in Section 5.3.1 is that syntactically equal operations can have different meanings (a temperature 'delta' parameter may indicate a positive or negative delta for example) or not be accessible at all times (e.g. in case of slow producers and fast consumers).

A second drawback of the lack of semantics is that dependencies and constraints often specify that components are incompatible, rather than why they are incompatible. From the point of view from an engineer that has to consider selecting a closely matching component, however, it is also very important to know why certain components exclude or depend on other components, or whether certain constraints are weak or strong, than just the fact that certain components cannot be selected as-is.

Solution and research topics: Relevant research issues, such as novel approaches to documentation, and literal programming, have been discussed by Bosch (2003) and the research issues on erosion of documentation.

5.7. Related Work

In the context of evolution, Bayer et al. (1999) and Clements and Norhtrop (2001) propose periodic scoping during domain engineering as solution for dealing with the continuous evolution of a product family. In this chapter, we have proposed the use of a feature scoping activity that continuously interacts with product derivation, to prevent non-optimal scoping decisions. Product family scoping is furthermore discussed in Kishi et al. (2002) and Schmid (2000).

The problems with implicit properties confirm problems regarding documentation in the context of software reuse discussed by Bosch (1999). Nonaka and Takeuchi (1995) provides a discussion on the process of externalization, i.e. converting tacit knowledge to documented or formalized knowledge. Literate programming was introduced in Knuth (1984), and addresses the erosion of documentation problem discussed in Section 5.6.

Several industrial case studies have been performed inside our group, e.g. on product instantiation (Bosch and Hogstrom, 2001) and evolution in software product families (Svahnberg and Bosch, 1999). Also outside our group, several case studies have been presented over the years, e.g. Ardis et al. (2001), Brownsword and Clements (1998), Clements et al. (2001). The contribution of this study is that it specifically identifies and analyses product derivation problems, whereas earlier case studies primarily focused on domain engineering.

5.8. Conclusion

In this chapter, we have presented the results of a case study on product derivation that involved two industrial organizations, i.e. Robert Bosch GmbH and Thales Nederland B.V. The case study results include a number of problems and causes we identified.

The discussions in this chapter focused on two core issues. The first issue is complexity. Although the assumption that variability improves the ability to select alternative functionality and thus the ease of deriving different product family members is easily made, the results of our case study suggest otherwise. At a certain point, each additional variation point leads to an increase in the cognitive complexity of the product family, and possibly complicates the derivation process. On the one hand, product derivation problems can be alleviated by product derivation methodologies that effectively deal with complexity, e.g. through hierarchical organization and first-class representation of variation points and dependencies. On the other hand, problems can be alleviated by addressing the main source behind complexity, i.e. evolution of variability. We have identified research issues related to evolution of variability, such as variability assessment techniques that assist software engineers in determining the optimal scoping decision during product derivation, as well as assessment techniques to determine the feasibility of removing obsolete variation points, and selecting a particular variability realization mechanism during domain engineering.

The second important issue is the large number of implicit properties (such as dependencies) of software assets. An approach to coping with this issue is to reactively document implicit properties that frequently result in mistakes. Problems can further be alleviated by addressing the sources of implicit properties, such as erosion of documentation, insufficient and over-explicit documentation, and the lack of semantics and behavior in specifications. Parts of these causes behind implicit properties confirm the problems related to documentation and reuse discussed by Bosch (1999). Research issues related to this topic can therefore also be found in that article.

Concluding, software product families have been successfully applied in industry. By studying and identifying product derivation problems, we have shown that there is room for improvement in the current practice (whether there is room in current theory is part of the next chapter). Solving the identified problems will help organizations in exploiting the full benefits of software product families. Our work therefore aims to define and validate methodologies that are practicable in industrial application and that address the product derivation problems discussed in this chapter.

Acknowledgments: This work was sponsored by the CONIPF project, under contract no. IST-2001-34438. We would like to thank the business unit Combat Systems at Thales Nederland B.V., as well as the business units at Robert Bosch GmbH. In particular, we would like to thank Paul Burghardt, Egbert de Ruyter (Thales), John MacGregor, Andreas Hein and Stefan Ferber (Bosch), for their valuable input.

5.9. References

Ardis, M., Daley, N., Hoffman, D., Siy, H., Weiss, D., 2000. Software Product Lines: A Case Study, Software – Practice and Experience, Vol. 30, No. 7, pp. 825–847.

Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.-M., 1999. PuLSE™: A Methodology to Develop Software Product Lines, Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99), pp. 122-131.

Bosch, J., 1999. Product-line architectures in industry: a case study, Proceedings of the 21st International Conference on Software Engineering, pp. 544-554.

Bosch, J., Högström, M., 2000. Product Instantiation in Software Product Lines: A Case Study, Second International Symposium on Generative and Component-Based Software Engineering, pp. 147-162

Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, H., Pohl, K., 2001. Variability Issues in Software Product Lines, Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), pp. 11–19.

Brownsword, L., Clements, P., 1996. A Case Study in Successful Product Line Development, Technical Report CMU/SEI-96-TR-016, Software Engineering Institute, Carnegie Mellon University.

Clements, P., Cohen, S., Donohoe, P., Northrop, L., 2001. Control Channel Toolkit: A Software Product Line Case Study, Technical Report CMU/SEI-2001-TR-030, Software Engineering Institute, Carnegie Mellon University.

Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns, SEI Series in Software Engineering, Addison-Wesley, ISBN: 0-201-70332-7.

Jacobson, I., Griss, M., Jonsson, P., 1997. Software Reuse. Architecture, Process and Organization for Business Success. Addison-Wesley, ISBN: 0-201-92476-5.

Javadoc, 2003. http://java.sun.com/j2se/javadoc/

Kishi, T., Noda, T., Katayama, T., 2002. A Method for Product Line Scoping Based on Decision-Making Framework, Proceedings of the Second Software Product Line Conference, pp. 348–365.

Kostoff, R. N., Schaller, R. R., 2001. Science and Technology Roadmaps, IEEE Transactions on Engineering Management, Vol. 48, no. 2, pp. 132-143.

Knuth, D.E., 1984. Literate programming, Computer Journal Vol. 27, pp. 97-111.

Nonaka, I., Takeuchi, H., 1995. The Knowledge-Creating Company: How Japanese companies create the dynasties of innovation, NewYork: Oxford University Press.

Schmid, K., 2000. Scoping Software Product Lines - An Analysis of an Emerging Technology, Proceedings of the First Software Product Line Conference, pp. 513–532.

Svahnberg, M., Bosch, J., 1999. Evolution in Software Product Lines: Two Cases, Journal of Software Maintenance, Vol. 11, No. 6, pp. 391-422.

Svahnberg, M., Gurp, J. van, Bosch, J., 2002. A Taxonomy of Variability Realization Techniques, ISSN: 1103-1581, Blekinge Institute of Technology, Sweden.

Chapter 6 Classifying Variability Modeling Techniques

Variability modeling plays an important role in variability management in software product families, especially during product derivation. In the past years, several variability modeling techniques have been developed, each using its own concepts to model the variability provided by a product family. The publications regarding these techniques were written from different viewpoints, use different examples, and rely on a different technical background. This chapter sheds light on the similarities and differences between five variability modeling techniques, by exemplifying the techniques with one running example, and classifying them using a framework of key characteristics for variability modeling. It furthermore discusses the relation between differences among those techniques, and the scope, size, and application domain of product families.

Based on	**Section numbers**
M. Sinnema, S. Deelstra, Classifying Variability Modeling Techniques, Elsevier Journal on Information and Software Technology, Vol. 49, pp. 717–739, July 2007.	All sections in this chapter

6.1. Introduction

Cost, quality, and time-to-market have been the main concerns in software engineering since the 1960s. Software product families (Bosch, 2000; Clements and Northrop, 2001; Jacobson et al, 1997) address these concerns by exploiting the similarities within a set of products. While developing components for a set of products enables effective reuse, it also complicates software development; the product family artifacts (such as, the architecture, reusable components, requirements, and test-cases) have to be able to deal with the product differences.

Variability management deals with these differences by handling the introduction, use, and evolution of variability, i.e. the ability of a system or artifact to be extended, changed, customized, or configured for use in a specific context. In this case, the context is the set of products in the product portfolio that are created during application engineering and the different environments these products are used in. The success of a product family is largely dependent on effective variability management (Bosch et al., 2001). It is a complicated task, however, that faces a number of challenges. These challenges originate, for example, from the large number of choices application engineers can make during product derivation (the numbers can run as high as ten thousands), and the complexity of the constraints between them (Bosch et al., 2001; Chapter 3).

Over the past years, several variability modeling techniques have been developed that are aimed to support variability management during product derivation, e.g. FODA (Kang et al., 1990), featureRSEB (Griss et al., 1998), Riebisch et al. (2004), Forfamel (Asikainen, 2004), RequiLine (Maßen and Lichter, 2002), Cardinality-Based Feature Modeling (CBFM) (Czarnecki et al., 2005), Maßen and Lichter (2004), Halmans and Pohl (2003), ConIPF (Hotz et al., 2006), CONSUL/Pure::Variants (Beuche et al., 2004; Pure Systems Website), GEARS (Krueger, 2002), Koalish (Asikainen et al., 2004), OVM (Pohl et al., 2005), VSL (Becker, 2003), van der Hoek (2004), and Gomaa and Webber, (2004). These techniques aim at representing the variability in a product family, in such a way that software engineers can manage variability more easily during product derivation. They have similarities and differences in terms of core aspects that are important when managing variability, i.e. in terms of modeling and in terms of the tools that support them. It are these differences that make each variability technique suitable for a particular situation. An important outstanding question, however, is which technique is most suited for which situation.

This chapter is a first step in answering this question, and its contribution is threefold. First, we provide an overview of variability modeling techniques, by showing how they model variability. Whereas other publications regarding these techniques are written from different viewpoints, use different examples, and rely on a different technical background, we use one example product family to illustrate the modeling concepts of all techniques. Second, we provide a detailed classification of these techniques. This classification is based on the characteristics that are common among variability modeling techniques, or that are based on issues that are encountered in practice. Finally, we discuss the relation between differences among these techniques, and the scope, size, and application domain of product families.

We divided this chapter as follows. First, we explain the structure of our classification framework by discussing related work, and outlining the concepts, activities and issues that are involved in management variability during product derivation (Section 6.2). Then, we illustrate the modeling techniques in Section 6.4 by means of a small example product family described in Section 6.3. We classify these techniques in Section 6.5 and 6.6, with the classification framework from Section 6.2. We discuss how well they match different types of product families in Section 6.7. We conclude our chapter in Section 6.8.

6.2. Background and Classification Framework

In product families, product development is separated in two phases, i.e. domain engineering and application engineering (Bosch, 2000; Clements and Northrop, 2001). Domain engineering involves creating and maintaining a set of reusable artifacts (e.g. a product family architecture, and a set of components). During application engineering, these reusable artifacts are used to build the products.

In both phases, variability management plays an important role. During the first phase, domain engineers introduce all kinds of variability mechanisms (e.g., parameterization or abstraction) in the software artifacts, so that application engineers can make specific choices (extension, change, customization, or configuration) that suit the required functionality and quality of the product they are creating.

On the one hand, variability contributes to the success of reuse, as variable artifacts are designed to operate in different circumstances. On the other hand, variability management also has to deal with a number of complications that arise during application engineering. First, not all choices in the product family artifacts can be made independent of each other. Choosing a particular component, for example, may require or exclude the presence of other components. In addition, a decision can influence multiple quality attributes simultaneously, while the precise impact of a decision on these quality attributes may not be clear until testing (For a more detailed discussion on this issue, see Part II). Finally, the number of choices and constraints can run into the ten thousands, which makes it difficult to maintain an overview of the effects of all choices (Bosch et al., 2001 and Chapter 3).

Combined, these complications can lead to situations where product derivation is a laborious task that is highly dependent on experts, and where a large number of iterations is required through product development. This is primarily due to issues such as, mistakes, not being able to find relevant information, and trial-and-error iterations to meet required quality attributes.

Variability modeling techniques have been developed to assist engineers in dealing with the complications of variability management. The idea behind these techniques is *representing* the variability provided by the product family artifacts, in such a way that it provides an increased overview and understanding of variability, and enables tools to take over some of the variability management tasks. In literature, many approaches can be found that deal with aspects of variability management, however. To avoid confusion as to which of those techniques adhere to our definition, we first discuss related work.

6.2.1. Related Work

The first distinction we make between variability modeling and other techniques is based on the difference between variability modeling and variability mechanism. As we mentioned above, variability modeling techniques model the variability that is provided by the product family artifacts, while variability mechanisms are commonly considered as ways to introduce or implement variability in those artifacts. Several of these mechanisms have been identified, such as conditional compilation, patterns, generative programming, macro programming, and aspect oriented programming (Bachmann and Bass, 2001; Clements and Northrop, 2001; Jacobson et al, 1997; Svahnberg et al., 2005; Zhang and Jarzabek, 2004). This means approaches such as GenVoca and Ahead by Batory and O'Malley (1992) and Batory et al. (2003), Frames by Basset (1987), XVCL by Zhang and Jarzabek (2004) and Aspect Oriented Programming by Kiczales et al. (1997) are not considered variability modeling techniques in this chapter.

There are some exceptions in literature regarding the distinction between modeling and mechanisms, however. Keepence and Mannion (1999), for example, speak of patterns to model variability. In our terminology, patterns would be a mechanism, so the approach discussed in Keepence and Mannion is also not considered a variability modeling technique in this chapter.

The second distinction focuses on the modeling part itself. In our view, a variability modeling technique has a well defined language for representing the variability. The approaches of Atkinson et al. (2000), Schmid et al. (2005)[1] and Coplien et al. (1998) do document variability, for example, but these are in the form of a table or section in a text document.

A large number of what we do consider as variability modeling techniques remains. First, there are different flavors of techniques that solely focus on modeling variability with features, such as FODA (Kang et al., 1990), featureRSEB (Griss et al., 1998), Riebisch et al. (2004), Forfamel (Asikainen, 2004), RequiLine (Maßen and Lichter, 2002), and CBFM (Czarnecki et al., 2005). Second, there are different flavors of variability modeling with use cases, such as von der Maßen and Lichter (2004), and Halmans and Pohl (2003). Finally, there are other techniques, such as ConIPF (Hotz et al., 2006), CONSUL/Pure::Variants (Beuche et al., 2004; Pure Systems Website), GEARS (Krueger, 2002), Koalish (Asikainen et al., 2004), OVM (Pohl et al., 2005), VSL (Becker, 2003), van der Hoek et al. (2004), and Gomaa and Webber (2004). To be able to provide a detailed classification, we describe and classify five of these techniques, i.e. VSL,

[1] Note that at the time of publication, we overlooked the paper by Schmid and John (2004), which describes a notation independent approach to modeling variability.

ConIPF, CBFM, Koalish and Pure::Variants. In Section 6.4, we motivate the selection of this particular set of techniques.

We are not the first to compare variability modeling techniques. Most articles in which we encounter comparisons, however, are focused on pointing out related work, or motivating the contribution of the technique proposed in those articles. The different extensions of FODA, for example, typically discuss the added value of the extension (Czarnecki et al., 2005; Maßen and Lichter, 2002; Riebisch et al., 2004). The discussions in literature are mainly focused on whether the techniques incorporate a tool, or which language constructs appear in one techniques, but not in the other. ConIPF also provides a high-level discussion on the added value to FODA, RequiLine, Koalish, CBFM, and GEARS. This discussion boils down to pointing out the fact that ConIPF models variability on the feature and component level, and explicitly linking the two, or claiming that ConIPF is able to specify more complex relations.

A comparison that is not focused on promoting their own technique is published in Bontemps et al. (2004), and Schobbens et al. (2006). In those articles, the authors formally compare different extensions of FODA, such as featureRSEB, and CBFM. They discuss what constitutes a valid formal feature model, and compare the approaches accordingly. They show that, although the extensions provide benefits in terms of succinctness, they are not less ambiguous or more expressive than FODA itself (in terms of the valid combinations of features they can specify).

In addition, two technical reports exist, i.e. Trigaux and Heymans (2003), and a German technical report by Lichter et al. (2003). The first report compares feature modeling and use cases to model variability in requirements. They use a number of points on which to compare the techniques, such as representation of common and variable parts, distinction between types of variability, representation of dependencies between variable parts, support for model evolution, understandability, and graphical representation. They furthermore apply all techniques to one example. Lichter et al. also provide examples of different feature modeling techniques, but the examples differ for each technique. In addition, they compare different feature modeling techniques by looking at the original goal of the techniques, and differences in modeling elements.

The classification in this chapter is not meant to promote one particular approach, not solely focused on variability in requirements, nor a formal comparison of the modeling languages. We define a classification framework that lists a number of essential aspects that a variability modeling technique should poses in order to be useful during product derivation, and classify five techniques according to this framework. We discuss the parts of this framework in the following section.

6.2.2. Framework Categories

When applying variability modeling techniques, three aspects are important to engineers. The first important aspect is what it is they exactly need to model, and what constructs are available to do so. Second, they care about how tools support them in creating and using these models. Finally, they need to know which steps they should take in using these models and tools, i.e. what processes are defined.

As only very few modeling approaches discuss which processes should be applied when using them (an exception being the ConIPF Methodology (Hotz et al., 2006)), we only use the first two aspects as main categories for our classification framework, i.e., modeling and tools. Both categories are divided into constituent parts (called characteristics). The selection of each characteristic is the result of insights gained from three sources. First, there is an existing body of knowledge regarding software product families and variability management, for example, Bosch (2000), Clements and Northrop (2001), and Weiss and Lai (1999). Second, we compared the underlying concepts of the variability modeling techniques to identify similarities and differences. Finally, a number of case studies and other articles identify a set of issues that should be addressed for variability management during product derivation, e.g., Bosch et al. (2001), Brownsword and Clements (1996), and Chapter 3. In other words, the characteristics of our classification framework reflect the important facets of variability modeling, and everyday practice. Below, we discuss the individual parts of this classification framework.

Modeling

The first and most obvious aspect in classifying variability modeling techniques is how variability information is represented. When pointing out specific characteristics we need to discuss, the first characteristic immediately follows from the essence of variability, i.e., how choices are modeled. Other important characteristics are how products are modeled, which abstractions are used to manage complexity, how the constraints and quality attributes are modeled, and how incompleteness and imprecision are addressed. We discuss these topics below.

Choices. Variability is all about choices that enable engineers to extend, change, customize, or configure product family artifacts for use in a specific context. These choices emerge during all phases of the derivation process, such as requirements elicitation, the configuration of the code, and testing. In our classification framework, the concept of choice refers both to how the ability to choose is modeled, as well as how the possible options that can be chosen are modeled.

Product models. Ultimately, decisions have to be made during product derivation, which translate the set of variable product family artifacts into a specific product. A product model refers to how the specific selection from the available options is represented in the variability modeling technique.

Abstraction. Next to the complexity as found in terms of the relations and quality attributes, software engineers also have to deal with complexity in terms of sheer numbers. A commonly used principle for simplification and focus in modeling techniques is providing several abstractions. Abstraction allows the engineers to structure the choices in such a way that only relevant choices can be shown to the engineers.

Formal constraints. Given the range of choices in the variability model, not all combinations of decisions for these choices will lead to a useful and consistent product. The restrictions on the decisions for choices are referred to as constraints in this chapter. An example of a constraint is that when a specific option is selected, another option has to be selected as well to get a consistent product. When those constraints are explicitly formalized in the variability model, consistency of the configuration at hand can be checked (automatically) without having to test the resulting software product.

Quality attributes. The Fifth characteristic is how quality attributes are modeled. One of the hardest things during product derivation is meeting the required quality attributes. As quality attributes are also one of the crucial aspects of a product, it is important to know how variability modeling techniques deal with them.

Support for incompleteness and imprecision. In an industrial setting, large parts of the knowledge about the software product family can be either non-existent, incomplete, or imprecise. If we look at quality attributes, for example, the precise relation between the choices and the impact on the value of quality attributes is often unknown. This makes it very hard or impossible to create a formal specification that accurately describes these relations. However, experts frequently are able to provide an educated guess, for example, based on the fact that they have derived products with similar combinations of options before, or know which option for a particular choice will lead to a better score compared to another option. With this characteristic, we therefore classify to what extent the variability modeling technique expects relations to be specified, and how the technique handles imprecision.

Tools

One of the important reasons to use variability models is that they are a basis for tools that assist engineers in the decision process during product development. In practice, the usefulness of a modeling technique is even measured through the accompanying tool-suite. In the context of variability modeling, tool-support can be divided in terms of creation and maintenance of a variability model (a task that belongs to the domain engineering level) on the one side, and the use of this model (both in terms of domain and application engineering) on the other side. When we look at these tasks and the tools that accompany the variability modeling techniques, there are a number of characteristics that are important for variability management.

Views. First, tools can provide a user interface that presents an overview of the variability in a product family. This overview consists of one or more views, where each view has a specific focus and purpose.

Active specification. A second characteristic is preventing inconsistency. Medvidovic and Taylor (2000) term this aspect ‘active specification,’ which means reducing the possible actions that can be taken when creating or maintaining a model. Medvidovic and Taylor distinguish between proactive and reactive active specification. Proactive specification means the tool suggests or prevents certain actions, while a tool only informs the user of inconsistencies in case of reactive specification.

Configuration guidance. Third, tools can provide guidance through the wide range of choices in the product family, e.g., by presenting the choices in a particular sequence. Guidance is important, as it may prevent inconsistencies, provides focus on difficult choices, or assures that inconsistencies are found early in the process. This guidance can be dynamically calculated based on properties of the model, such as, the number of constraints between choices, or statically defined as a strategy.

Inference. With an inference engine, tools can automatically take decisions based on constraints and previous decisions. An inference engine typically consists of three main elements (Wikipedia Website, 2006), i.e., applying the known rules, determining the order in which rules have to be applied, and maintaining consistency.

Effectuation. Finally, the decisions that are taken during product development should result in an actual product. We refer to the mapping of the decisions to actual product family artifacts as the effectuation of a product. Examples of tasks in this effectuation part are the generation of Makefiles, setting compile flags, copying static libraries and generating start-up configuration files. Tools can assist the engineer by automatically performing these effectuation tasks based on the decision made in the model.

6.3. Running Example: Parking

Before we present our classification of modeling techniques in Section 6.5, we introduce the different techniques in Section 6.4. To get a feel for how the different modeling techniques model variability, however, we first present an example in this section. This example is based on a small product family we came across at one of our industrial partners. It is used in Section 6.4 to show how the variability model of this family would look like in the different modeling techniques. Please note that the only purpose of this small example case is to illustrate the variability modeling techniques, and that we by no means intended it to represent any part of the classification framework.

The example involves a product family in the domain of license plate recognition on handhelds, such as PDAs and smartphones. The products in this family are software applications that are used on parking lots to check whether the parked cars have a valid parking license. The basic idea is that the operator takes pictures of parked cars with his mobile device. The software on his device automatically recognizes the license plate and validates this license plate against a black-white list (a list specifying who does not or who does own a valid parking license). This validation is either done by sending the plate via WLAN or GPRS to a central server and receiving back the status, or by checking it against a daily-updated local list on the PDA itself. As shown in Figure 18, a white car with license plate "52-GS-TT" appears to have a valid parking license, as "Parking license" is reported back from the server.

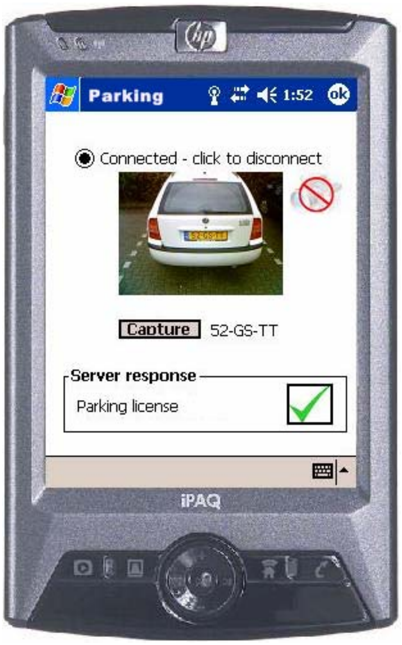


Figure 18. A HP iPaq 3715 running the Parking application. In the center of the display the live video stream from the camera is shown in the view-port. In the bottom of the screen, the response from the server is shown.

6.3.1. Design

The functionality described above is realized in the architecture of the Parking product family and its implementation in the components. Figure 19 presents the logical view of the product family architecture, which consists of five components. It shows that the *Parking Application* runs on an *Operating System* and uses a *Camera Interface* and a hardware specific *Display Interface*. These interfaces allow the Parking Application to use the display and camera of the mobile device, respectively. Throughout this chapter we refer to these components as `a-ParkingApplication`, `a-OperatingSystem`, `a-DisplayInterface`, `a-CameraInterface`, `a-WirelessInterface` and `a-Device`.

In this example we use prefixes to categorize the entities in this Parking example. The “a-“ prefix used above identifies the architectural components. The “c-“ prefix identifies choices, the “s-“ prefix identifies system properties and finally the “r-“ prefix identifies the constraints. The last three types of entities are described in Section 6.3.2 and 6.3.3.

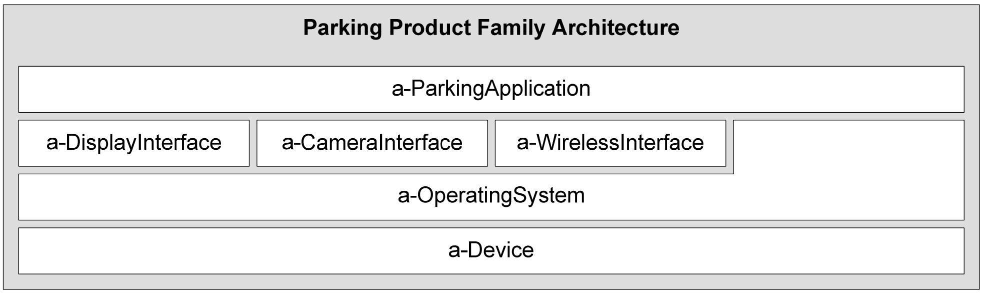


Figure 19. Logical View of the Product Family Architecture.

The implementation of the architecture of the Parking Product Family is written in the C++ programming language. The Parking Application is implemented in such a way that it can be compiled to a Symbian as well as a Windows CE library. Each hardware manufacturer supplies a specific implementation of the `a-CameraInterface` component as a static C++ library. These variants are identified by the device name suffix, i.e. `a-CameraInterface_iPaq`, `a-CameraInterface_Symbol` and `a-CameraInterface_P900`. The generic `a-DisplayInterface` and `a-WirelessInterface` components can be used for all three devices. Each product contains the `a-DisplayInterface` component. The `a-WirelessInterface` is optional in the architecture and supports communication via GPRS as well as WLAN. During the development of a Parking product, the three interface libraries are linked together with the `a-ParkingApplication` library to one executable. The `a-OperatingSystem` component on which this executable is running is also supplied by the hardware manufacturer.

6.3.2. Choices

Products from the Parking product family are delivered to customers in different application domains. These customers impose different requirements on the products that are ordered. The Parking product family provides a wide range of variability to be able to deliver those different products. As the purpose of this running example is just to illustrate the presented techniques, however, we have limited the variability of the Parking product family in this example to only seven choices, i.e. the validation method (`c-Validation`), the device (`c-Device`), the operation system (`c-OperatingSystem`), the implementation variant of the interface to the camera (`c-CameraInterface`), the presence of a wireless interface (`c-WirelessInterface`), the dimensions of the viewport (`c-ViewPortSize`) and the capture resolution (`c-CaptureResolution`). These choices are summarized in Table 5. The commonalities between the names of the component entities in Table 5 and the choice entities in Figure 19 might be confusing. Therefore, we use the prefixes to show the difference between the components themselves (e.g. `a-Device`) and the *choice* between variants of these components (e.g. `c-Device`).

Table 5. Choices and options in the simplified Parking Example. This table presents for each choice, a name (first column), a description (second column), a set of alternative options (third column) and indicates whether the choice is visible to the user of the system (fourth column).

Choice	Description	Options	User-visible
`c-Validation`	The validation method used by the PDA	Server-side Local	Yes
`c-Device`	The type of handheld device the Parking application is running on.	HP iPaq 3715 (`iPaq`) Symbol MC50 (`Symbol`) Sony-Ericsson P900 (`P900`)	Yes
`c-OperatingSystem`	The operating system the application is running on	Windows CE (`WindowsCE`) Symbian (`Symbian`)	Yes
`c-CameraInterface`	The choice between the different variants of the camera interface.	a-CameraInterface_iPaq a-CameraInterface_Symbol a-CameraInterface_P900	No
`c-WirelessInterface`	The choice of including the wireless interface in the architecture.	Present Absent	No
`c-ViewPortSize`	The dimensions of the live video stream in the application window.	80 × 60 120 × 100 300 × 220 etc.	Yes
`c-CaptureResolution`	The resolution of the pictures captured by the device.	200 × 120 640 × 400 1024 × 768 etc.	Yes

Table 6. Technical details of the devices in the Parking Example

c-Device	Camera resolution	Screen size	Memory	Supported OS	Wireless
iPaq	1280 × 960	240 × 320	56MB	`WindowsCE`	WLAN
Symbol	1140 × 880	320 × 320	64MB	`WindowsCE`	WLAN
P900	640 × 480	206 × 320	48MB	`Symbian`	GPRS

6.3.3. Constraints

In this small example, products cannot contain all combinations of options for the seven choices. The possible set of choices is limited by a few constraints. First, the device (`c-Device`) should support the operating system (`c-OperatingSystem`) that has been selected. Second, the camera interface implementation (`c-CameraInterface`) should match the device that is used. Third, the camera of the device that is used to take the pictures (`c-Device`), should support the resolution (`c-CaptureResolution`) required to capture the pictures. Fourth, the size of the view-port (`c-ViewPortSize`) should be supported by the camera of the device and should not exceed the dimensions of the screen. Fifth, when Server-side license plate validation is used (`c-Validation`), the wireless interface component should be present (`c-WirelessInterface`). Table 6 presents the data required to check whether these three constraints are violated or not.

As all knowledge required to validate these five constraints is available, they are easy to validate. There are, however, three important properties of a derived Parking product that impose three additional constraints on choices in the product family. These properties are the memory consumption (`s-Memory`), the average processing time (`s-ProcessingTime`) and the recognition rate (`s-RecognitionRate`), and are summarized in Table 7. Following from the `s-Memory` property, products should not require more memory than available on the device. Additionally, customer requirements on the system properties `s-RecognitionRate` and `s-ProcessingTime` may limit the possible configurations. The knowledge about these three constraints is far from exact; some heuristics that are known to the engineers are given in Table 8. Note that, as a result, these constraints are much harder to validate than the four described in the previous paragraph. An overview of all eight constraints is shown in Table 9.

Table 7. Three properties of products derived from the Parking Example.

System property	Description
`s-Memory`	The amount of memory (in MB) required to run the application.
`s-ProcessingTime`	The average time (in milliseconds) it takes to process (i.e. recognize and validate the license plate) one picture.
`s-RecognitionRate`	The percentage of license plates that are successfully recognized.

Table 8. The effect of the choices on the system properties. For this example, the values of the system properties are determined by feeding a fixed set of 1000 pictures into the application

Choice	Effect on system properties
`c-Validation`	Server-side validation requires more processing time (`s-ProcessingTime`) than local validation.
`c-Device`	The performance (influencing the `s-ProcessingTime`) and amount of memory (restricting the `s-Memory`) varies between these devices.
`c-OperatingSystem`	Since there's a one-to-one mapping of operating systems on devices, there's no additional effect of this choice on the system properties.
`c-CameraInterface`	No effect on the system properties.
`c-WirelessInterface`	The presence of a wireless interface requires extra memory (`s-Memory`).
`c-ViewPortSize`	A larger view-port size requires more memory (`s-Memory`).
`c-CaptureResolution`	A higher resolution implies a higher recognition rate (`s-RecognitionRate`), but requires more memory (`s-Memory`) and processing time (`s-ProcessingTime`).

Table 9. The six constraints in the Parking product family example. These constraints include the constraints that are related to the system properties in Table 7.

Constraint	Description
`r-OperatingSystem`	The hardware should support the selected operating system[2].
`r-CameraInterface`	The `a-CameraInterface` implementation for the selected hardware should be used.
`r-CaptureResolution`	The camera should support the capture resolution.
`r-ViewPortSize`	The camera and the display should support the selected view-port size.
`r-Validation`	Server-side validation of the recognized license plates requires the wireless interface to be present,
`r-Memory`	The memory consumption should not exceed available memory on device.
`r-RecognitionRate`	The recognition rate of the product should exceed the recognition rate required by its customer.

[2] This constraint might sound too trivial to mention as each as device only support one operating system. However, it allows to easily illustrate how constraints are modeled in the next section.

`r-ProcessingTime`	The average processing time of the product should not exceed the average processing time required by its customer.

6.4. Variability Modeling Techniques

In the overview on related work in Section 6.2.1 we listed several variability modeling techniques, developed to support engineers in managing their product family. To be able to discuss the classification of variability modeling techniques in detail, we have restricted the number of techniques we classify. We motivate this selection below.

CBFM (Czarnecki et al., 2005), featureRSEB (Griss et al., 1998), Riebisch et al. (2004), Forfamel (Asikainen, 2004) and RequiLine (Massen and Lichter, 2002) are all based on FODA (Kang et al., 1990) feature modeling. As CBFM is formalized, and supported by mature tool support from which we could easily get an evaluation copy, we selected this technique to represent the other flavors of FODA. The same holds for the variation point modeling techniques Gomaa and Webber (2004), OVM (Pohl et al., 2005) and VSL (Becker, 2003), from which we selected VSL based on the absence of tooling for the other two techniques. Koalish (Asikainen et al., 2004) and van der Hoek (2004) integrate variability into an existing architecture modeling language, i.e. Koala (Ommering et al., 2000) and xADL (Dashofy, et al., 2001), respectively. Based on the large amount of specification details, we included Koalish in the classification framework to represent these two techniques. Pure::Variants (Beuche et al., 2004; Pure Systems Website), ConIPF (Hotz et al., 2006) and GEARS (Krueger, 2002) are also mature techniques, different to the techniques above. These techniques should therefore be part of the classification framework as well.

Due to unavailability of specification language and tooling, however, we could not evaluate the latter properly. We therefore left GEARS out of this classification framework. The variability modeling techniques of von der Maßen and Lichter (2004) and Halmans and Pohl (2003) model variability in use cases and focus on variability in the problem domain. As our classification aims at the modeling of the provided variability in the solution domain, these two are also excluded from this assessment.

Summarized, we selected five variability modeling techniques for our classification: CBFM, VSL, ConIPF, Pure::Variants, and Koalish. For each technique, we briefly introduce the basic concepts of the modeling languages and show how the Parking example from the previous section can be modeled using these concepts. The purpose of this section is to get insight in the techniques, a thorough classification with the framework from Section 6.2 is provided in Section 6.5.

6.4.1. Variability Specification Language (VSL)

The first of the five techniques we classify, i.e. the Variability Specification Language (VSL) (Becker, 2003), is an technique to modeling variability that distinguishes between variability on the specification and on the realization level. The variability on the specification level refers to choices demanded by customers and is expressed by "Variabilities." On the realization level, "Variation Points" indicate the places in the asset base that implement these required choices. They describe the actions that should be taken when a particular option is chosen. *Implements* relations between Static Variation Points and Variabilities describe how the Variation Points should be configured based on the decisions for the Variabilities. The Variation Points are separated into two sets, i.e., Dynamic Variation Points (referring to runtime variability) and Static Variation Points (referring to pre-runtime variability).

Figure 20 shows how the Parking product family from Section 6.3 can be modeled using the concepts in the meta-model of VSL. The structure of this figure matches the structure of the original Figure 20 in the publication on VSL (Becker, 2003), which means the top-right box describes the specification level, the bottom-right box describes the realization level and the left box describes the software assets that build up the product family. The legend in Figure 20 shows how the graphical elements map onto the concepts in the VSL.

The five Variabilities on the specification level (the shaded boxes in Figure 20) correspond to the five user-visible choices described in Table 5, i.e. `c-Validation`, `c-Device`, `c-OperatingSystem`, `c-ViewPortSize` and `c-CaptureResolution`. The options for these choices are modeled by the Variant entities that are part of the Range of the Variability entities. Dependency entities in VSL allow the specification of formal constraints e.g. *requires* and *excludes* relations, on the selection of one or more Variant entities. Therefore, only the constraints `r-Validation`, `r-OperatingSystem`, `r-CaptureResolution` and `r-ViewPortSize` from Table 9 can be modeled by VSL, and the constraints `r-Memory`, `r-RecognitionRate` and `r-ProcessingTime` remain tacit as expert knowledge.

The left part of Figure 20 contains the compositional structure of the software application. This part of the model contains three different types of entities. The StaticAsset and GenericAsset entities represent the overall structure without variability aspects, and correspond to the entities in Figure 19. Variability in this structure is expressed by DerivedAsset entities. These entities represent alternatives for the GenericAssets, e.g. the three options of the `c-CameraInterface` choice and the optionally in `c-WirelessInterface` in Table 5.

The choice between the three camera interface implementations is represented by the Static Variation Point `c-CameraInterface` on the realization level. In the VSL variability model, this Static Variation Point is contained in the corresponding GenericAsset (`a-CameraInterface`). The *implements* relation between `c-CameraInterface` and `c-Device` contains the `r-CameraInterface` constraint from Table 9 and describes which `a-CameraInterface` implementation should be used based on the selected `c-Device` variant. Similarly, the optionally of `a-WirelessInterface` is represented by the Static Variation Point `c-WirelessInterface`. The *implements* relation between this `c-WirelessInterface` and the Variability `c-Validation` specifies the `r-Validation` constraint. As the Parking example does not include runtime variability, no Dynamic Variation Points are contained in Figure 20.

6.4.2. ConIPF

The basic idea behind the ConIPF Methodology (Hotz et al., 2006) is to capture all knowledge required to successfully derive products in such a way that products are configured on the level of Features. Based on the selected Features and relations to hard- and software components, its associated tool suite infers which hard- and software components have to be selected and how their parameters should be set. For this tool support, ConIPF has tailored the configuration tool suite of Encoway GmbH (Encoway website). This tailored tool suite is based on KBuild (for building knowledge models) and KConfig (for deriving individual products).

The ConIPF variability model contains Asset entities with Relations between them. ConIPF distinguishes between several types of Assets, e.g. *context* entities, *features*, *hardware* entities and *software* entities. The Relations are also specialized into several types, e.g. *has-context*, *has-features*, *has-hardware*, *has-subsystems* and *has-software*. These Relations represent the hierarchical structure on the assets. Choices within this hierarchy are enabled by allowing alternative and optional assets connected through these relations for which the engineer makes a decision during product derivation. The individual Asset entities can furthermore be enriched with Parameters and Restrictions. These restrictions are used to specify constraints on choices.

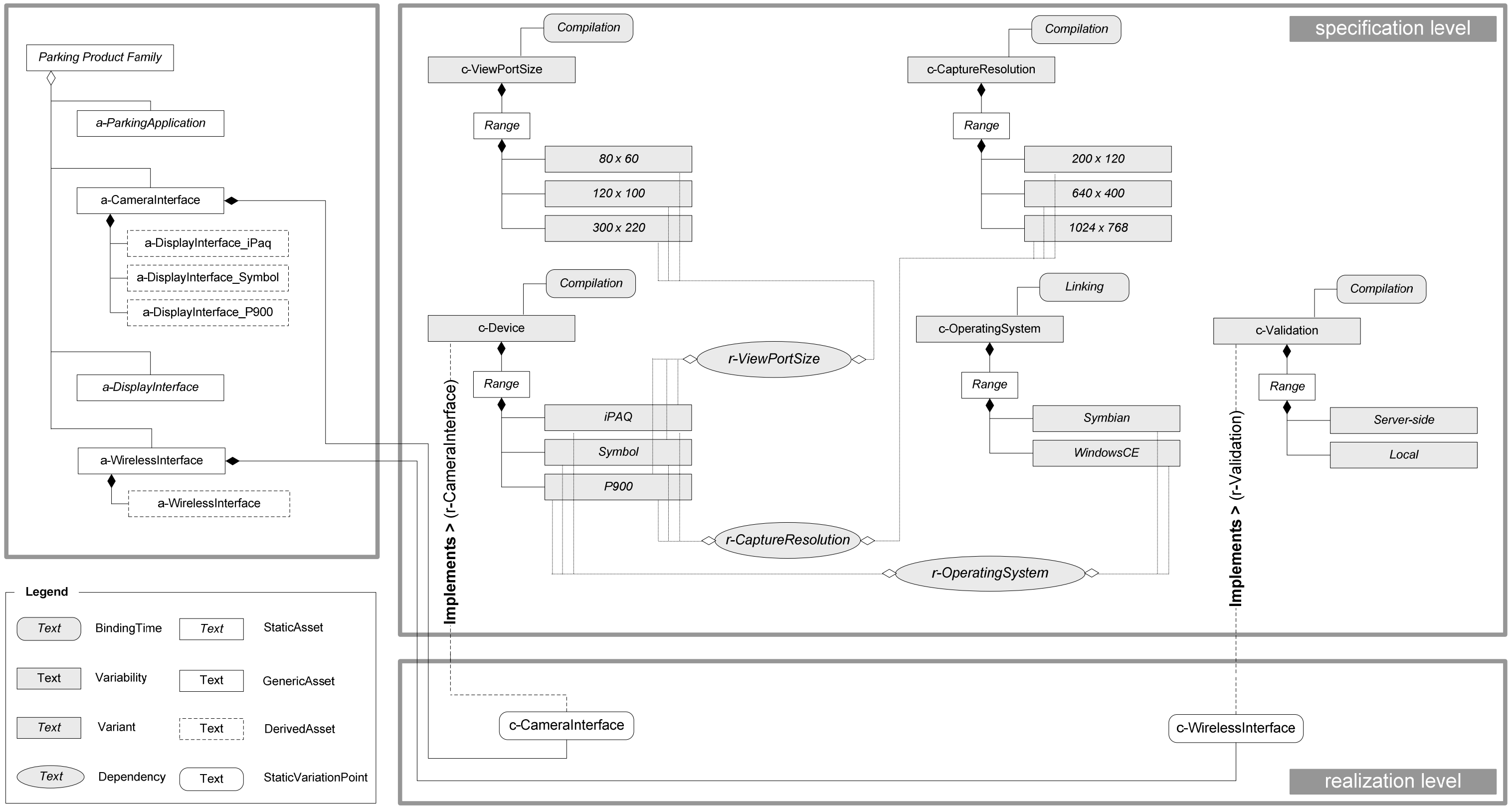


Figure 20. The Parking product family modeled in the Variability Specification Language. The top-right box (specification level) contains the user-visible choices from Table 5, the left box contains the software application structure from Figure 19 and the right-bottom box (realization level) specifies the variability in these assets that implement the user-visible choices on the specification level.

In Figure 21 we show how the Parking example would be modeled in the ConIPF variability model. Its top level element ParkingApplication (referred to as a *product* in ConIPF) is the aggregate of all Assets in the model. It contains different types of Relations to other assets contained in the ParkingApplication. The idea is that, based on manual decisions on the choices in the left part of this model (the features), the right part of this model (the hard- and software components) can be configured automatically. Therefore, the left part contains the user-visible choices from Table 5 and the system properties from Table 7. The right part represents the component structure from Figure 19 and includes the variability in this structure. We describe this left and right part in more detail below.

First, the three system properties from Table 7 are contained as *context* entities in the ParkingApplication via a *has-context* relation. It represents the external requirements on non-functional aspects of a Parking product. Second, the user-visible choices from Table 5 are modeled as the *features* `c-Validation`, `c-ViewPortSize`, `c-CaptureResolution`, `c-Device` and `c-OperatingSystem`. They are contained in the ParkingApplication via *has-features* relations. Finally, the parameters of `c-ViewPortSize`, and `c-CaptureResolution` represent the corresponding options in Table 5. The dashed line around the alternatives shows that at most one of them is allowed in one product.

The right half of Figure 21 represents the structure of the hard- and software components and corresponds to the architectural components in Figure 19. The *hardware* `a-Device` is connected to the *product* ParkingApplication via a *has-hardware* relation and the subsystem `a-OperatingSystem` is connected via a *has-subsystems* relation. The *subsystem* `a-ParkingSystem` is connected to the ParkingApplication via a *has-subsystems* relation and contains the three *software* entities `a-CameraInterface`, `a-DisplayInterface` and `a-WirelessInterface`.

Although the graphical representation of the ConIPF variability model gives a good overview of the entities in the product family, it does not give insight in the parameters and the restrictions. Therefore, we also included the textual representation of three entities in the modeling language of ConIPF (Hotz et al., 2006). This modeling language is called AMPL (i.e. Asset Model for Product Lines). Table 10 shows an example of how AMPL represents a *product*, a *feature* and a *restriction* entity.

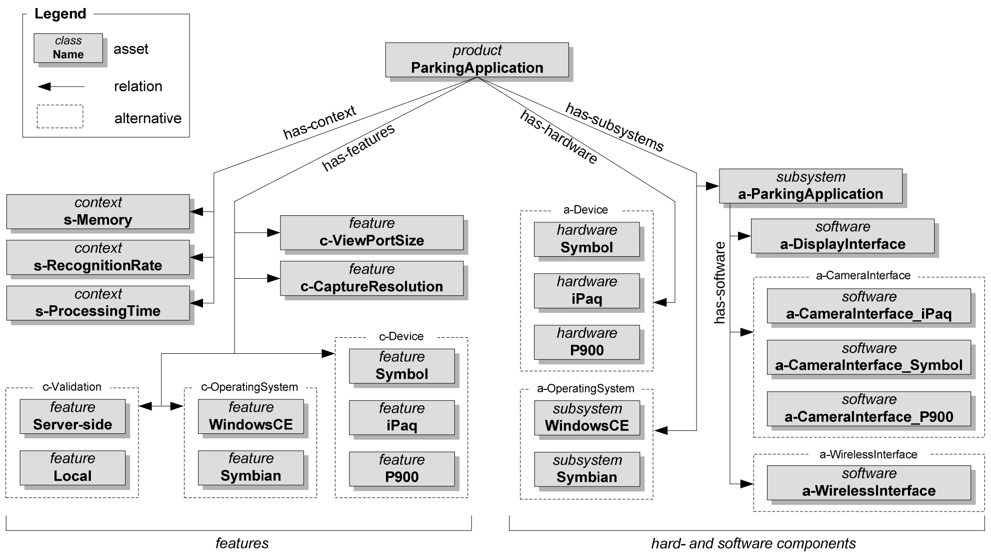


Figure 21. The Parking product family modeled in the ConIPF Methodology. During product derivation features are selected (left part) and tools automatically determine the required hard-and software components based on these features (right part).

The AMPL specification of the Assets in the first and second row of Table 10, starts with the Asset type, followed by its name and its composition in terms of relations. The relations consist of the relation type (e.g. `has-context`) and the related parts. To accommodate for variability, the parts are preceded by one of the keywords *alternative* and *mandatory*. As we show in the description of the feature entity `c-ViewPortSize`, Assets can also contain variability in terms of parameters, which are attributes that are specified by a name and a valid range. The third row of Table 10 shows how *restrictions* are represented, i.e. first the class, followed by its name, the scope, and a function in the scope that should evaluate to true.

6.4.3. Cardinality-Based Feature Modeling (CBFM)

Based on FODA (Kang et al., 1990), Cardinality-Based Feature Modeling (Czarnecki et al., 2005) addresses the challenges of product family engineering by modeling the variability solely in terms of choices in Features. In this technique, Features denote any functional or non-functional characteristic in the product family artifacts. To allow for abstraction, Features are hierarchically organized in a Feature tree. Each Feature can have one attribute that represents a numeric or textual property of the Feature.

Table 10. The textual representation of three different assets in ConIPF.

Entity	Type	AMPL representation
ParkingApplication	Product	**Product** **Name:** `ParkingApplication` **Relations:** **Composition:** `has-context` **Parts:** **mandatory** `s-Memory` **mandatory** `s-RecognitionRate` **mandatory** `s-ProcessingTime` **Relations:** **Composition:** `has-features` **Parts:** **mandatory** `c-ViewPortSize` **mandatory** `c-CaptureResolution` **alternative** `WindowsCE, Symbian` **alternative** `iPaq, Symbol, P900` **alternative** `Server-side, Local` **Relations:** **Composition:** `has-hardware` **Parts:** **alternative** `iPaq, Symbol, P900` **Relations:** **Composition:** `has-subsystem` **Parts:** **alternative** `WindowsCE, Symbian` **mandatory** `a-ParkingApplication`
`c-ViewPortSize`	Feature	**Feature** **Name:** `c-ViewPortSize` **Parameters:** **Name:** `Width`, **Value**: `[0, 10000]` **Name:** `Height`, **Value**: `[0, 10000]`

`r-ViewPortSize` and `r-CaptureResolution` for iPaq device.	Restriction	**Restriction**
		Name: `iPaq_Restrictions`
		Exists: `ParkingApplication,`
		`s-Memory` **context-of** `ParkingApplication,`
		`c-ViewPortSize` **feature-of** `ParkingApplication,`
		`c-CaptureResolution` **feature-of** `ParkingApplication,`
		`iPaq` **hardware-of** `ParkingApplication`
		Function:
		`s-Memory.MemoryConsumption <= 56,`
		`c-ViewPortSize.Width <= 240,`
		`c-ViewPortSize.Height <= 320,`
		`c-CaptureResolution.Width <= 1280,`
		`c-CaptureResolution.Height <= 960`

The variability in the product family is represented by Feature cardinality, Feature groups, and Feature attributes. Feature cardinality means that in the feature tree of a product family one can specify the number of times the Feature may occur in one product, e.g., a cardinality of [0..1] represents the choice for a Feature to occur once or not at all in a product. Feature groups express a choice between multiple sub-features. Feature attributes can also represent choices, as the actual value of Feature attributes can remain unspecified until a product is configured. Accordingly, the configuration process breaks down into specifying the cardinality of features, selecting features from feature groups and specifying the open attribute values.

Figure 22 shows the feature model of the Parking example from Section 6.3. The "ParkingSystem" Feature forms the root of the tree, containing eight mandatory sub-Features. The sub-Features `c-Validation`, `c-Device`, `c-OperatingSystem`, `c-CameraInterface`, `c-WirelessInterace`, `c-ViewPortSize` and `c-CaptureResolution` represent the choices in Table 5, respectively. The `c-Device` Feature contains the Feature group of `P900`, `Symbol` and `iPaq`, which correspond to the options for the choice `c-Device` in Table 5. The sub-Features of the Features in this Feature group contain information about the size of the display and the resolution of the camera of the device in their Feature attributes.

The Feature attributes are used to specify constraints on the whole Feature model, e.g. to restrict the acceptable values for the "Height" and the "Width" of the c-ViewPortSize feature. CBFM allows for formally specifying these constraints. They are formulated using XPath 2.0 (3WC Website, 2005) expressions that evaluate to true or false. Table 9 presents the constraints we formulated in the Parking example. This table is included just to illustrate the type of constraints that can be formulated. For a full description of constraint specification in this technique we refer to Antkiewicz and Czarnecki (2004).

FeaturePlugin (Antkiewicz, 2004) personal is a plug-in for Eclipse that was developed to model variability according to the concept of features described above. This tool features an interface for modeling the product family and configuring products.

6.4.4. Koalish

Koalish (Asikainen et al., 2004) is a variability modeling language that enables variability modeling in product families that are built using Koala (Ommering et al., 2000). Koala is a software modeling technique that uses components and interfaces to specify the logical structure of a software system, without variability. The model specifies which components exist in the software system and whether a component is part of another component. The interfaces specify how components interact with other components. For this purpose, components can contain required and provided interfaces. A connection between two components is referred to as the binding between a required interface of one component and a provided interface of the other component. For more details on Koala we refer to (Ommering et al., 2000).

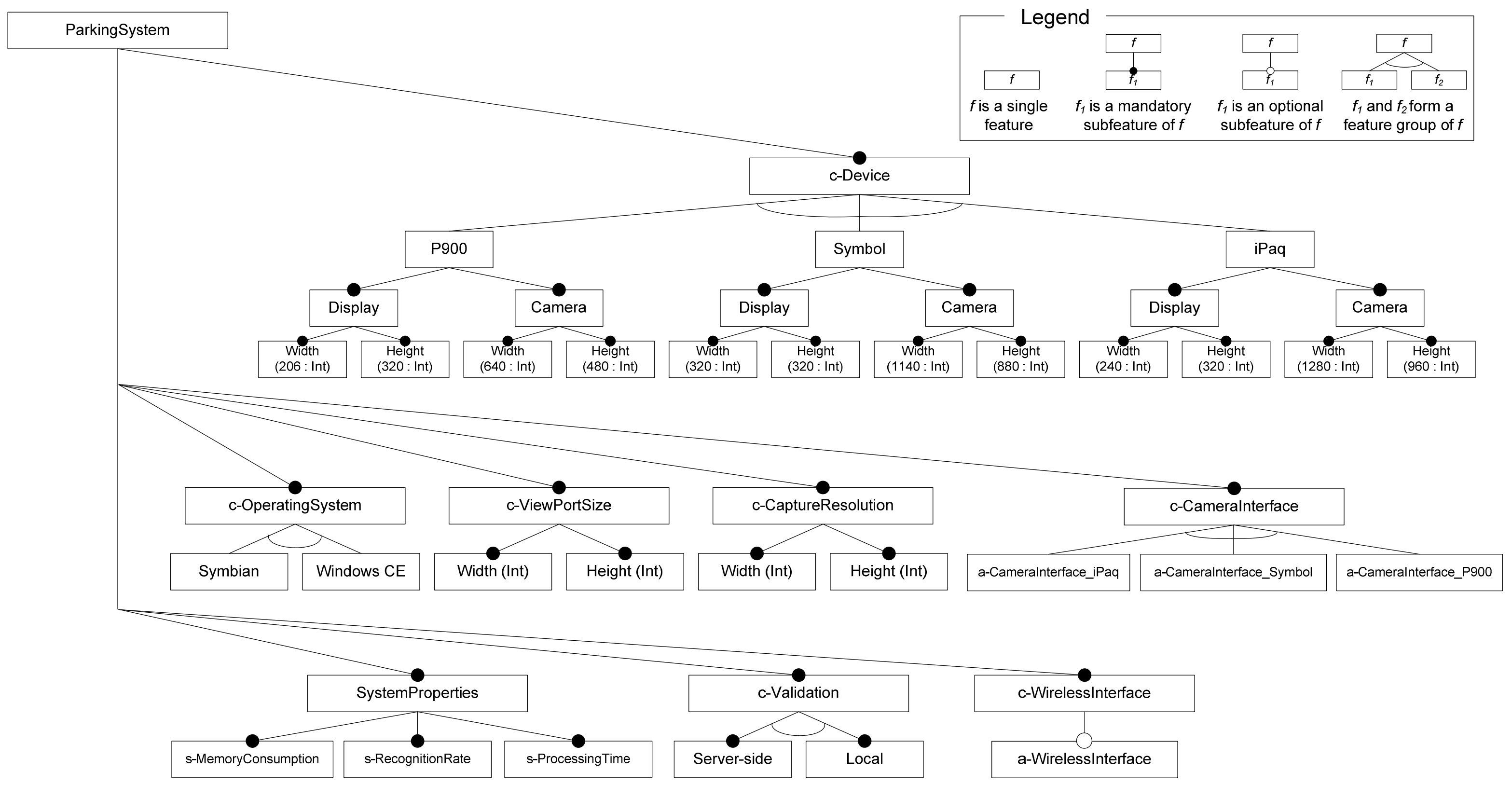


Figure 22. The CBFM feature model of the Parking product family.

Table 11. The constraints in the CBFM feature model of the Parking product family.

Formal Specification	Description
if (//c-Device/P900) then (//c-OperatingSystem/Symbian) else true()	r-OperatingSystem The operating system should match the device.
if (//c-Device/Symbol) then (//c-OperatingSystem/WindowsCE) else true()	
if (//c-Device/iPaq) then (//c-OperatingSystem/WindowsCE) else true()	
if (//c-Device/iPaq) then (//c-CameraInterface/a-CameraInterface_iPaq) else true()	r-CameraInterface The camera-interface implementation should match the device.
if (//c-Device/Symbol) then (//c-CameraInterface/a-CameraInterface_Symbol) else true()	
if (//c-Device/P900) then (//c-CameraInterface/a-CameraInterface_P900) else true()	
if (//c-Validation/Server-side) then (//c-WirelessInterface/a-WirelessInterface) else true()	r-Validation
some $x in //c-Device satisfies ($x//Display/Width/@value ge //c-ViewPortSize/Width/@value)	r-ViewPortSize The ViewPort should fit in the display of the device
some $x in //c-Device satisfies ($x//Display/Height/@value ge //c-ViewPortSize/Height/@value)	
some $x in //c-Device satisfies ($x//Camera/Width/@value ge //c-CaptureResolution/Width/@value)	r-CaptureResolution The Capture Resolution should not exceed the resolution of the camera.
some $x in //c-Device satisfies ($x//Camera/Height/@value ge //c-CaptureResolution/Height/@value)	

In Figure 23 we show how the software of a product from the Parking product family can be modeled with Koala. Note that this figure does not show any choices, as Koala models do not support variability. A Parking system in this context contains four subcomponents, i.e. the a-ParkingApplication, the a-DisplayInterface, the a-CameraInterface and the a-WirelessInterface. The latter three provide the interfaces *iuD* of type *UseDisplay*, *iuC* of type *UseCamera* and *iuW* of type *UseWireless*, respectively. These provided interfaces are bound to the required interfaces *iuC*, *iuD* and *iuW* of the Parking Application. All four subcomponents contain a software module that implements the interfaces of the component.

The basic idea behind Koalish is extending the concepts of Koala with variability, e.g. by modeling optional and alternative Koala components. During the configuration process a Koala model is derived from this Koalish model. As Koala focuses on the architecture and implementation level and does not abstract to features, Koalish does not allow for abstraction to features either. Koalish is furthermore text-based; it does not feature a graphical representation. More details on the configuration process and the available tool support, i.e. the Kumbang Configurator, can be found in Myllärniemi et al. (2005).

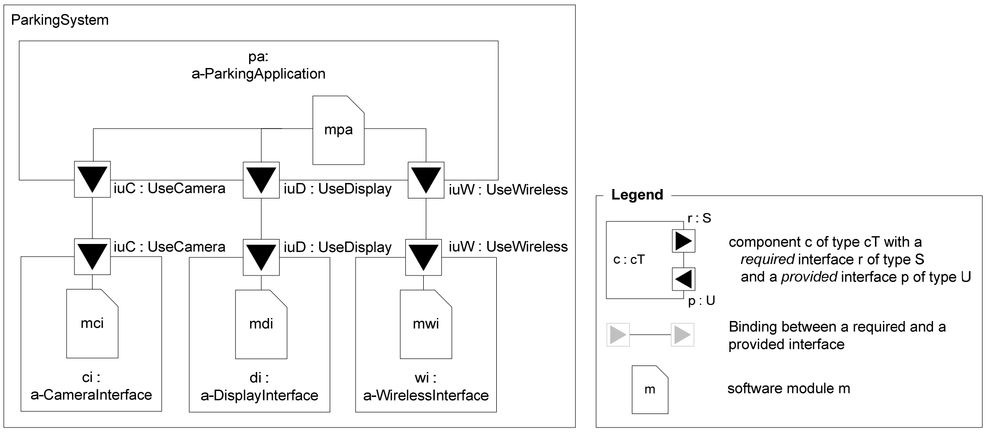


Figure 23. A generic Parking product modeled with Koala. This model does not contain variability.

Figure 24 illustrates how Koalish can be used to model the Parking Example from Section 6.3. It shows how the ParkingSystem component (the `a-ParkingApplication` together with the `a-DisplayInterface`, `a-CameraInterface` and `a-WirelessInterface`) is textually represented in the Koalish language. Its structure is a one to one mapping from Figure 23, where variability has been incorporated. In the ParkingSystem component, the choices are specified in the "contains" part, where alternatives are summarized between braces. In Figure 24, the ParkingSystem component contains four subcomponents, e.g. *pa* (which is either a `a-ParkingApplication_WindowsCE` or a `a-ParkingApplication_Symbian`). The "constraints" part specifies three constraints on the choices in the "contains" part, i.e. which combinations of options are valid. The first constraint refers to the `r-OperatingSystem` constraint combined with the logical structure in Figure 19. It basically says that when *ci* (the camera interface) is an instance of `a-CameraInterface_iPaq`, then *di* (the display interface) should be an instance of `a-DisplayInterface_iPaq` and *pa* (the parking application) should be an instance of `a-ParkingApplication_WindowsCE`.

```
component ParkingSystem {
  contains
    component (a-ParkingApplication_WindowsCE,a-
ParkingApplication_Symbian) pa;
    component DisplayInterface di;
    component (a-CameraInterface_iPaq,a-CameraInterface_Symbol,a-
CameraInterface_P900) ci;
    component (a-WirelessInterface) wi[0-1];
  constraints
    ci instance of a-CameraInterface_iPaq => pa instance of a-
ParkingApplication_WindowsCE;
    ci instance of a-CameraInterface_Symbol => pa instance of a-
ParkingApplication_WindowsCE;
    ci instance of a-CameraInterface_P900 => pa instance of a-
ParkingApplication_Symbian;
}
```

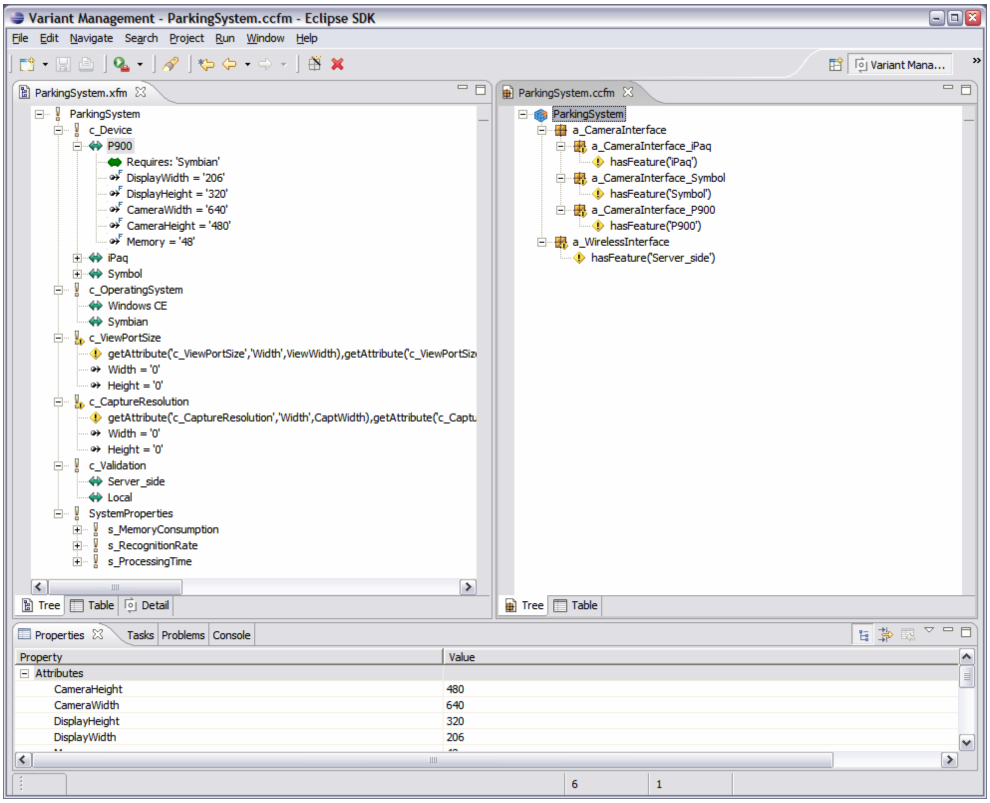
Figure 24. An example of how Koalish incorporates variability in a Koala component.

6.4.5. Pure::Variants

Pure::Variants (Pure Systems Website) is a tool-suite that is based on the CONSUL approach (Beuche et al., 2004). It is similar to configuration principles from AI techniques for the configuration of physical products. Like these AI techniques, it uses an object-model with specialization (is-a) and aggregation (part-of) relations to model variability.

The Pure::Variants Eclipse Plug-in adds a Variant Management Perspective to Eclipse. This Perspective provides different views for modeling and configuration. The models are visualized by tables and tree structures, where icons are used to inform the user about types, relationships, and evaluation problems, for example (see Figure 25).

Variability modeling in Pure::Variants is organized around four types of models, i.e. *feature models*, *variant description models*, *family models* and *result models*. The feature and family models consist of entities called Elements and Attributes. These entities are objects that are guarded by Restrictions. Restrictions are either simple requires or conflicts restrictions, but it is also possible to specify more complex Prolog restrictions. If none of the Restrictions of an object evaluates to true, the object cannot be part of a configuration of the feature or family model (the variant description model, or result model, respectively). Elements furthermore contain Relations that specify 1:n relations between Elements, such as, *requires*, *requiredFor*, *conflicts*, *recommends*, or *discourages*. Relations and restrictions thus formally specify constraints.


Variant Management - ParkingSystem.ccfm - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Variant Mana...
ParkingSystem.xfm
ParkingSystem.ccfm
Tree
Table
Detail
Properties
Tasks
Problems
Console

```
ParkingSystem
  c_Device
    P900
      Requires: 'Symbian'
      DisplayWidth = '206'
      DisplayHeight = '320'
      CameraWidth = '640'
      CameraHeight = '480'
      Memory = '48'
    iPaq
    Symbol
  c_OperatingSystem
    Windows CE
    Symbian
  c_ViewPortSize
    getAttribute('c_ViewPortSize','Width',ViewWidth),getAttribute('c_ViewPortSiz
    Width = '0'
    Height = '0'
  c_CaptureResolution
    getAttribute('c_CaptureResolution','Width',CaptWidth),getAttribute('c_Captu
    Width = '0'
    Height = '0'
  c_Validation
    Server_side
    Local
  SystemProperties
    s_MemoryConsumption
    s_RecognitionRate
    s_ProcessingTime
```

```
ParkingSystem
  a_CameraInterface
    a_CameraInterface_iPaq
      hasFeature('iPaq')
    a_CameraInterface_Symbol
      hasFeature('Symbol')
    a_CameraInterface_P900
      hasFeature('P900')
  a_WirelessInterface
    hasFeature('Server_side')
```

Property	Value
Attributes	
CameraHeight	480
CameraWidth	640
DisplayHeight	320
DisplayWidth	206

6 1

Figure 25. Screenshot of the variability model of the Parking Example Product Family in Pure::Variants. The left pane in this figure shows the Feature Model, while the right pane shows the Family Model.

Feature Models: Figure 25 shows a screenshot of the Parking Example modeled with Pure::Variants. The left-most tree-structure in this figure represents the Parking Example Feature Model. The Feature Models in Pure::Variants specify FODA-like features in terms of Feature Elements and Feature Attributes. Pure::Variants supports four types of features, i.e. *mandatory*, *optional* (0..1), *alternative* (1 out of n), and *or* (i..j out of n). The Features that are typed *alternative*, and *or* are grouped in different feature groups, in such a way that one parent Feature can only have one *alternative*, and one *or* feature-group. The Feature Attributes allow expressing domain features in terms of values. These values can be pre-defined, pre-calculated or set during configuration.

The first way in which choices are modeled using Pure::Variants are thus feature types, and configurable feature attributes. As depicted in Figure 25, for example, the feature `c_Device` represents the choice for the device on which the Parking application will run (see Table 5). The three children, i.e. `P900`, `Symbol`, and `iPaq`, correspond to the options for this choice. Each of these children has pre-defined

Attributes that denote the display size, camera resolution, and amount of memory of the handheld device.

Variant Description Models. A variant description model in Pure::Variants describes which Features and Attributes from the feature models are included in a configuration, along with the value for the Feature Attributes. The inclusion of Features is controlled by the feature type and relations. A mandatory Feature is included when a parent Feature is included. The other Features are selected by a user, or their selection is inferred by the Pure::Variants tool.

A configuration from a Feature Model to a Variant Description Model is invalid if not at least one restriction evaluates to true, or when at least one relation evaluates to false. When the feature P900 is included in the Variant Description Model, for example, the model is invalid if it does not contain the feature Symbian, as the relation that is part of P900 specifies that it requires Symbian. See Table 12 for more examples of how Pure::Variants models constraints.

Table 12. Constraints in Pure::Variants. This table shows three examples of how constraints in the Parking product family are modeled in Pure::Variants.

Formal Specification	**Description**
requires('Symbian')	Example of a relation of the P900 feature that specifies the selected operating system should be Symbian.
hasFeature('iPaq')	Example of a restriction on the `iPaq a_CameraInterface`. It specifies the `a_CameraInterface_iPaq` component will be included only if the iPaq feature is selected.
getAttribute('c_ViewPortSize','Width',ViewWidth), getAttribute('c_ViewPortSize'','Height',ViewHeight), getAllSelectedChildren('c_Device',[SelectedDevice]), getAttribute(SelectedDevice,'DisplayWidth',DisWidth), getAttribute(SelectedDevice,'DisplayHeight',DisHeight), (DisWidth>=ViewWidth),(DisHeight>=ViewHeight)	Example of a restriction that specifies the ViewPort should fit in the Display. The restriction on the width and height are combined in one restriction as the Pure::Variants configuration would be considered correct even if the restrictions would be split up and only one of the dimensions would fall within the specified limit. This restriction is part of the `c_ViewPortSize` feature, which also has two attributes called Width and Height.
getAttribute('c_CaptureResolution','Width',CaptWidth), getAttribute('c_CaptureResolution','Height',CaptHeight), getAllSelectedChildren('c_Device',[SelectedDevice]), getAttribute(SelectedDevice,'CameraWidth',CamWidth), getAttribute(SelectedDevice,'CameraHeight',CamHeight), (CamWidth>=CaptWidth),(CamHeight>=CaptHeight)	Example of a restriction that specifies the Capture Resolution should not exceed the resolution of the camera.

Family Models: The family models on the other hand, describe the family in terms of software (architectural) elements. The main Elements in family models consist of Parts that are grouped into Components. Each Part is associated with any number of SourceElements that determine the way to generate the source code for the specified Part (e.g., through file copy or adding pre-compiler directives). Parts refer to programming language elements such as classes, objects, flags, or variables, but this model is open to introduction of other types. Next to relations and restrictions between elements in the family model, the relations and restrictions are used to link the feature models to the family models.

The right tree structure in Figure 25 specifies such a family model for our Parking Example. The family model of the Parking Example in Figure 25 only contains the variable components on architectural level, i.e. the `a_CameraInterface` components.

Result Models: The result models specify which of the Elements that are described by the family models are included in a product. Unlike Features, the Elements in a Family Model are not typed *mandatory*, *optional*, *alternative*, or *or*, and it is not possible to select elements or set attributes as a user. Rather, the elements are included in a result model only if a parent element is included, and at least one restriction evaluates to true. A configuration is invalid if the at least one relation evaluates to false.

Choices in family models are thus implicit. In the family model, all entities are possible options, where evaluation of restrictions determines whether an entity is present in the result model. To illustrate this point, each sub-component of the `a_CameraInterface` component of the Parking Example has a restriction of the form hasFeature(<Feature>). For the `a_CamperaInterface_iPaq`, for example, this restriction is hasFeature('iPaq'), which means that the iPaq a_CameraInterface component is only part of the configuration if the Feature `iPaq` is part of the variant description model.

6.5. Classification of the Variability Modeling Techniques: Modeling

The brief descriptions in the previous section already suggest that VSL, ConIPF, CBMF, Koalish, and Pure::Variants have commonalities and differences. But what are these commonalities and differences exactly, and how does that make them suitable for different types of product families? To provide a first answer to this question, we focus on the first half in Sections 6.5 and 6.6; explicitly describing the commonalities and differences using the structure of our classification framework (see also Section 6.2). We summarize this classification, and discuss the suitability with respect to different types of product families in Section 6.7.

The first category of our classification framework consists of comparing how the variability is represented in the variability models. The important topics regarding this category are choices, product representation, abstraction, formal constraints, quality attributes, and support for incomplete knowledge.

6.5.1. Choices

The techniques basically employ two different ways to model choices, i.e. as *multiplicity in the structure* or as *choice models*. The main difference between these approaches is that, where the main concept in the first approach is the structure of the software and features or components are the first-class entities, the main concept of the models in the second approach is variability, and choices are the first-class entities.

Multiplicity in structure

Although their specific implementation differs, most of the variability modeling techniques use the *multiplicity in the structure* approach to model variability. The main entities in their models are features and/or components, and aggregation relations between these entities specify its structure. Variability is enabled by allowing the entities in the models to be optional and multiple, by providing alternative entities in the trees, or a combination of both. This is usually realized by aggregate entities that specify for each of their sub-entities how many times they are allowed in one product, e.g., zero or one times, one or more times, etc.

The techniques that use this approach are Pure::Variants, Koalish, ConIPF, and the CBFM. The main concept of these approaches are the combination of the Feature Models and the Family Models, the aggregation of the Koala components, the combination of a feature tree and an artifact tree, and a feature tree, respectively.

Choice models

The main concepts in techniques that use the choice models are the choices. In these models, relations between choices specify how choices depend on each other. Although these choices can refer to specific components or features (like the approach above), the choices and constraints in choice models are orthogonal to the structure behind the features and components. Choice entities are typically associated with a number of option entities of which zero or more (depending on the type of choice) can be selected for a product.

There is one variability modeling technique that uses this approach, i.e. the VSL The choices in VSL models are the union of the variability and the variation point entities (referring to choices on the feature level and the component level, respectively).

6.5.2. Products

The variability modeling techniques also represent the products derived from their variability model in different ways. Some represent them as stand-alone entities, separately from the variability model. Other techniques represent them by the decisions that have been made for the choices in the variability model.

Stand-alone entities

The product models of techniques that represent them as stand-alone entities have no link to the choice concepts in the product family model anymore. As a result, changes to the product family model will not affect the product models. On the one hand, product models will not benefit from future improvements to the product family model. On the other hand, product models will remain consistent in themselves.

Koalish is the only technique that uses this approach to represent the decisions that engineers make for the products. Products are represented by Koala models and have no explicit relation to the original choices in the Koalish models.

Decision models

The second way of representing products, i.e., by defining the products in the context of the choices in the variability model, enables a strong link between the products and the original variability model. In contrast to stand-alone product models, these models can benefit from future improvements to the product family model. Future changes, however, might be out of the context of the decision for the original products. As a result, these changes can invalidate existing products. Except for Koalish, all techniques from Section 6.4 use this approach to model their products.

6.5.3. Abstraction

To manage the large number of choices and constraints, the variability modeling techniques use abstraction by introducing a *hierarchy* in the variability and/or by modeling variability in *multiple layers*.

Pure::Variants, and ConIPF use hierarchy as well as multiple abstraction layers for abstraction in their variability model. Pure::Variants defines a hierarchical tree structure on the entities (features and components) in the variability model. Multiple layers have been realized by allowing multiple trees (Feature Models and Family Models) to build up one variability model. ConIPF variability models contain several layers, such as the Context, the Feature, the Hardware and the Artifact tree (the parts of the ParkingSystem in Figure 21). The part-of relations within these layers specify the hierarchical ordering between assets, e.g. the Software assets as part of the `a-ParkingApplication` in Figure 21.

The variability model of VSL does not allow the specification of a hierarchical ordering of the Variability entities, and, although they are related to Asset entities, there is neither a hierarchical ordering of Variation Point entities. Instead, abstraction in VSL is realized by using multiple layers. In that respect, Variability entities on the specification level abstract from the variability specified by the Variation Point entities on the realization level.

The other way around, Koalish and CBFM only incorporate hierarchy in their variability models. In Koalish models, the Koala components are hierarchically ordered. This hierarchy is specified in the *contains* part of the component descriptions (line 3, 4 and 5 in Figure 24).

6.5.4. Formal constraints

The way variability modeling techniques formally define constraints differs considerably. Its expressiveness ranges from just include and exclude relations to advanced algebraic expressions. VSL is the technique that restricts the type of constraints to include and exclude relations between two or more options. In VSL, these relations are specified between Variant entities on the specification level. The other techniques also allow for some kind of algebraic expressions, specified in either a technique-specific or in a more widely adopted constraint language.

Pure::Variants and CBFM employ the approach of tailoring an existing constraint language for usage in their own variability model. This facilitates the process of introducing the technique in an organization, as engineers are more likely to be familiar with it and the organization can use existing documentation and tooling. Pure::Variants constraints are basically Prolog rules on the entities in a Feature Tree. The CBFM constraints are XPath expressions on the entities in the Feature tree. Although the usage of Prolog and XPath is focused on specifying logical expressions over one or more entities, they are also capable of specifying simple mathematical expressions.

The other two techniques use a technique-specific language, which may hamper its introduction in an organization. Although the syntax of the constraints in these techniques differs, its expressiveness is similar. In ConIPF, constraints are specified as *functions* that calculate the validity of a configuration. These functions can contain mathematical functions with the parameter names of the Assets as variables, exemplified in the last row of Table 7. Besides the *instance* of constructs we showed in Figure 17, constraints in Koalish can also contain the basic arithmetic and logical operators. The complete syntax of constraints in the last two techniques is described in Hotz et al. (2006) and Asikainen et al. (2004), respectively.

6.5.5. Quality Attributes

Despite the fact that quality attributes are such an important aspect for product development, they get surprisingly little attention in the variability modeling techniques. Most ignore them, or only briefly mention something about it. For VSL, for example, it is mentioned that especially quality attributes make variability management difficult, but how quality attributes are modeled by VSL is not discussed. CBFM and ConIPF, on the other hand, only mention that non-functional aspects are modeled as features, and the context tree, respectively. Later, however, ConIPF also suggests that non-functional aspects can be represented by restrictions. Both techniques, however, lack a discussion and example of how this works exactly.

6.5.6. Support for Incomplete Knowledge and Imprecision

This characteristic of variability modeling techniques answers the question whether and how the technique deals with the fact that (in particular in an industrial setting) knowledge is unavailable, imprecise and incomplete. The result of the Koala derivation process is a full Koala model that can be compiled to C code. It therefore is aimed at having the whole architecture formalized in a complete variability model.

The idea behind ConIPF and Pure::Variants is that, based on the selection of features, the tooling automatically selects and configures the right components for the product. This idea requires that all relations between the choices at the feature layer and other layers are fully specified. Admittedly, ConIPF does allow the user to select options at lower levels of abstraction, but in principle both techniques are aimed a variability models that completely cover the variability provided by a (part of) a system at all abstraction layers. Koalish, ConIPF, Pure::Variants, and VSL, do not allow specifying imprecise relations.

6.6. Classification of the Variability Modeling Techniques: Tools

Tasks that are automated may reduce costs and increase the efficiency of product family engineering. Therefore, provided the modeling characteristics we discussed in Section 6.5, tools can support the engineers in managing their product family models. In the Tools category, we show whether and how the tools of each of the techniques support the characteristics we presented in Section 6.2.2.

6.6.1. Views

The support for views in the different tools varies from providing no separate views at all to the support for an almost unlimited range of views. As VSL is not supported by a modeling or configuration tool, and Koalish only provides the Kumbang Configurator, VSL, and Koalish do not support separate views at all. The other techniques are supported by tooling that at least provided two basic views, i.e., one view for building and maintaining the variability model, and another view to configure individual products.

The ConIPF tooling features two textual views in the form of the web-based tools KBuild and KConfig, for modeling and configuration, respectively. For CBFM and Pure::Variants, similar views are realized by two graphical views on the tree structure in their Eclipse plug-ins. In addition, Pure::Variant provides table views that list all possible or selected entities..

6.6.2. Active Specification

Whether the techniques actively reduce the possible actions that can be taken when creating or maintaining a model, is primarily related to the fact whether the techniques are supported by specific tooling in the first place. VSL, for example, is not supported by a specific editor to manipulate the Variation points, Variabilities, and Relations. It relies on generic XML editors to create, change and (re-)organize entities and therefore classifies as providing no active specification. Similar, Koalish requires a text-editor to manipulate the Koalish models and therefore neither supports active specification. Efficiently modeling the choices and relations with these techniques therefore requires precision and a lot of experience with the modeling language.

Pure::Variants and CBFM both provide an Eclipse plug-in that allows the engineer to graphically build the variability models. The plug-ins allow for proactive creation, change, and (re-) organization of the choices in the tree structures. They differ in terms of how they handle constraints, however. CBFM requires manual text-editing to describe Relations using XPath expressions, but checks the consistency of relations reactively once a relation is specified. Pure::Variants supports the definition of Relations between Entities proactively by only allowing the user to select existing Entities from tables. Restrictions, however, can also be defined manually, and Pure::Variants does not check the consistency of Relations and Restrictions, for example, once Entities are removed from the model.

ConIPF provides an editor for proactively manipulating the choices as well as constraints in the product family artifacts. It uses a third-party variability modeling tool that provides a web-interface. The tool prevents inconsistency, for example, by preventing the specification of constraints between choices that do not exist, and preventing the deletion of choices when there are still constraints referring to them.

6.6.3. Configuration Guidance

As we explained in Section 6.2.2, configuration guidance is concerned with the suggestions that engineers receive regarding the order in which the various decisions should be made. Only one technique provided tooling with configuration guidance, i.e. ConIPF. The technique provides means to suggest the order of decisions dynamically as well as statically.

Based on the constraints in the ConIPF variability model, KConfig dynamically determines which choices should be made first. This dynamic strategy can be overridden with procedural knowledge, a static strategy that specifies a particular order. The configuration tool KConfig presents these strategies to the engineer when applicable.

Engineers that use the configuration view of Pure::Variants, Koalish, or CBFM have to develop their own strategy for making decisions in the tree-shaped variability model, e.g. breath first or depth first. The tooling does not provide hints with respect to which order should be followed to efficiently derive the product at hand. As the tool support of VSL does not feature a configuration view at all, it does not provide Configuration Guidance either.

6.6.4. Inference

In addition to the guidance during the configuration process, an inference engine in a configuration interface can check the consistency of the configuration at hand. Except in the VSL, which has no configuration interface, this consistency checking is supported by the tooling of all modeling techniques. The Pure::Variants and CBFM Eclipse plug-ins both continuously check the configuration for consistency and give feedback to the user whenever an inconsistency is detected.

Inference engines can also be used to automatically infer decisions based on the constraints and decisions that are already taken during configuration. Pure::Variants provides an inference engine to determine which Elements should be selected from the family models. Note that such an inference engine still gives only feedback on consistency after a decision is made and it does not prevent the engineer from making decisions that would make the configuration inconsistent.

The prevention of making such decisions requires a more sophisticated inference engine and is provided with the KConfig tool of ConIPF, and the Kumbang Configurator tool of Koalish. In addition to consistency checking and inferring choices, these tools also reduce the possible decisions in the configuration interface to the set of decisions that have been calculated to result in a valid configuration.

6.6.5. Effectuation

The product models that are created during configuration are basically a description of which product family artifacts should be present in the final application, and how they should be configured. Therefore, this description still has to be *effectuated* in the product family artifacts. This effectuation step results in an application that can be tested or shipped to the customer. Four of the five techniques we discussed in Section 6.4 provide some form of tool support for this effectuation step.

In Pure::Variants, an XML transformation engine uses the variant description model and result model to create a product. It provides standard support for file-based operations based on the element types, with special support for C/C++ variability mechanisms, like specifying pre-processor directives and the creation of other C/C++ language constructs. The transformation engine can furthermore be extended with user specific transformation modules.

Descriptions of products in VSL are referred to as profiles. These profiles are XML descriptions of how the variation points in the product family artifacts should be configured. The effectuation step in VSL therefore boils down to processing the VSL descriptions in the artifacts based on the profile of the product. Although this concept enables effectuation of several different artifacts types, at this point, the supported types are XML documents and JScript sources.

The output of KConfig, the configuration interface of ConIPF, is an XML description that contains the structure of the final product. Although KBuild is not provided with generic effectuation support, companies can easily develop their specific tooling that transforms this XML description into the required realization in the product family artifacts.

The output of the Kumbang Configurator for Koalish models is a Koala description of the software product. The open source Koala-C tool (Koala website) transforms a Koala component into its realization in the C programming language. The result is a combination of a Makefile, C-files and H-files that can be compiled to an executable. Similar to ConIPF, for usage in a specific context, additional tools can be developed to transform Koala models into other types of artifacts.

In contrast to the other five techniques, the tooling for CBFM only goes as far as the configuration of the feature models. Czarnecki and Eisenecker (2000), however, do acknowledge the link between the CBFM feature models and artifacts.

6.7. Discussion

The classification we presented in Section 6.5 and 6.6 is summarized in Table 13. It briefly states whether and how the variability modeling techniques provide support for the eleven different characteristics.

From the differences identified in Table 13, we can conclude that the applicability of the techniques strongly depends on the properties of the product family at hand. One example of such a property is the supported programming language, which enforces obvious requirements on the Effectuation of choices. Other important properties are the scope of reuse of the product family, the size of the product family, and the stability of its application domain. As the relation between these properties and our classification is less obvious, we discuss these last three.

6.7.1. Scope of reuse and size of the product family

In Chapter 3, we identified four different scopes of reuse to classify software product families. This scope of reuse denotes to which extent the commonalities between related products are exploited and ranges from a *standardized infrastructure* to a *configurable product family*. The asset base of product families with one of the first two scopes, the *standardized infrastructure* and the *platform*, only contain components that are common to all products, and variability is mainly realized by product specific adaptations to these components. It therefore will probably only pay off to externalize the variability knowledge from small parts of the asset base.

The asset bases of software product families that are classified as a *software product line* not only provide the functionality common to all products, but also the functionality that is shared by a sufficiently large subset of product family members. Functionality specific to one or a few products is still developed in product specific artifacts.

The suitability of the techniques for the first three scopes of reuse primarily depends on the required support for incompleteness and imprecision for the family or parts of the family. For most product families we encountered in practice, and that employ a platform or software product line, the knowledge related to the impact of choices on quality attributes is typically something that involves a great deal of incompleteness and imprecision. Judging from Table 13, this suggests none of the techniques in our classification is capable of dealing with this aspect.

The *configurable product family* is the situation where the organization possesses a collection of shared artifacts that captures almost all common and different characteristics of the product family members, i.e., a configurable asset base. In general, new products are constructed from a subset of those artifacts and require no product specific deviations. The maturity level of a configurable product family is typically such that most relations in the family are well understood. This allows the formalization of choices and constraints in such a way that tools can automatically calculate a consistent combination of required components and parameter settings. The approaches that do offer inference include Pure::Variants, Koalish, ConIPF, and CBFM (see Table 13).

Table 13. The classification of five variability modeling techniques

	Approaches Characteristics	VSL	ConIPF	CBFM	Koalish	Pure::Variants
Model	Choices	Choice model (Variabilities and Variation Points)	Multiplicity in structure (Feature and Artifact tree)	Multiplicity in structure (Feature tree)	Multiplicity in structure (Koala components)	Multiplicity in structure (Feature and Family model)
	Products	Decision Model	Decision Model	Decision Model	Stand-alone entity	Decision Model
	Abstraction	Multiple Layers	Hierarchy and Multiple Layers	Hierarchy	Hierarchy	Hierarchy and Multiple Layers
	Formal Constraints	Include/exclude (technique specific)	Algebraic expressions (technique specific)	Algebraic expressions (XPath)	Algebraic expressions (technique specific)	Algebraic expressions (Prolog)
	Quality Attributes	Not modeled	Mentioned (Context entities, restrictions)	Mentioned (Feature entities)	Not modeled	Not modeled
	Support for Incompleteness and Imprecision	Requires precise specifications	Aimed at completeness, requires precise specifications	Requires precise specifications	Requires precise specifications	Aimed at completeness, requires precise specifications
Tooling	Multiple Views	Not supported by tooling	2 (modeling & configuration)	2 (modeling & configuration)	1 (configuration)	4 (modeling & configuration)
	Active Specification	Not supported by tooling	Proactive	Reactive specification of constraints	None	Proactive is possible, no consistency checking
	Configuration Guidance	Not supported by tooling	Static and dynamic	None	None	None
	Inference Engine	Not supported by tooling	Consistency, decision making, prevention	Consistency	Consistency, decision making, prevention	Consistency, decision making
	Effectuation	Operations on XML documents and Jscript sources.	None provided.	None provided.	Generation of C applications.	File-based operations with special support for C and C++.

When only looking at the size of product families, the suitability primarily depends on characteristics such as abstraction, multiple views, configuration guidance, and inference. All these characteristics help reducing the complexity, and amount of manual labor. The approaches that address most of these characteristics are Pure::Variants, Koalish, and ConIPF (see Table 13).

6.7.2. Stability of the application domain

When considering the adoption of a variability modeling technique, the third important property of a product family is the stability of its application domain. Depending on, for example, the level op innovation, maturity of the organization, or customer needs, the product family may continuously change, or change slowly. These changes need to be reflected in the variability model, or else these models will be rendered useless.

While we didn't incorporate characteristics that are specifically focused on evolution in our classification framework - because most approaches do not address this topic - there are for example tool characteristics that help in dealing with evolution. Active specification is one of these characteristics, as it helps engineers in keeping the model consistent while making changes. As VSL or Koalish do not support active specification, this suggests that they are more suitable for stable application domains (see also Table 13).

6.8. Conclusion

Variability modeling is recognized as the key approach to successful management of industrial software product families. In the past few years, several variability modeling techniques have been developed, each one using its own concepts to capture the variability provided by reusable artifacts. The publications regarding these techniques are written from different viewpoints, use different examples, and rely on a different technical background. As a result, it is hard to compare them and choose from these techniques.

The main contribution of this chapter is a detailed classification of five variability modeling techniques, i.e. CBFM, VSL, ConIPF, Pure::Variants, and Koalish. These five techniques form a representing subset of variability modeling techniques that have been developed during the last ten years. We briefly *introduced* each technique, *illustrated* each technique by means of an example case, and *assessed* each technique with respect to several characteristics of modeling and tooling.

If we look back to our discussion on variability in Section 6.2, and our classification in Section 6.5, there are three important issues that can be identified.

First, variability management is a complicated task, where many choices and constraints are involved. While we discussed the relations between the differences of the variability modeling techniques and the scope, size, and application domain of product families, this discussion was based on the properties of the techniques and the types, and not on the application in an industrial setting. Most publications on the variability modeling techniques do claim that they have been tested on one or more cases. These case studies, however, all seem to involve very small configurable product families. The scalability (see also Section 6.5.3 Abstraction) and suitability of the techniques with respect to other types of product families therefore remains questionable until more extensive case studies are performed.

Second, as we indicated in Section 6.2, most variability modeling techniques lack a description of a process, which is required before they can be successfully deployed. In particular, such a process should provide the engineer with the means to address the extraction of variability from the existing product family, and the evolution of such a model in response to change.

Finally, most approaches are based on a principle that requires a fully formalized variability model. In practice however, many product families are situated in a context where the level of required formalization cannot be achieved (think, for example, of the difficulty of formalizing quality attributes). This is not the result of any incompetence of the engineers, but due to the fact that these product families involve complex (technical) systems in a frequently changing domain. Both change and complexity make it too hard, or too expensive to formalize all variability information.

6.9. Bibliography

Antkiewicz, M., Czarnecki, K., 2004. FeaturePlugin: Feature Modeling Plug-In for Eclipse, Proceedings of the 2004 OOPSLA workshop on eclipse technology exchange, pp. 67-72.

Asikainen, T., 2004. Modelling Methods for Managing Variability of Configurable Software Product Families, Licentiate Thesis of Science in Technology at Helsinki University of Technology.

Asikainen, T., Soininen, T., Männistö, T., 2004. 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.

Atkinson, C., Bayer, J., Muthig, D., 2000. Component-based product line development: the KobrA Approach, Proceedings of the First Conference on Software Product Lines: Experience and Research Directions, p.289-309.

Bachmann, F., Bass, L., 2001. Managing Variability in Software Architectures, Proceedings of the Symposium on Software Reusability (SSR`01).

Basset, P. G., 1987. Frame-Based Software Engineering, IEEE Software, Vol. 4, No. 4, pp. 9-16.

Batory, D., O'Malley, S., 1992. The Design and Implementation of Hierarchical Software Systems with Reusable Components, ACM Transactions on Software Engineering and Methodology, 1(4), pp. 355-398.

Batory, D., Sarvela, J., Rauschmayer, A., 2003. Scaling Step-Wise Refinement, Proceedings of the 25th International Conference on Software Engineering, pp. 187-197.

Becker, M., 2003. Towards a General Model of Variability in Product Families, Proceedings of the 1st Workshop on Software Variability Management, Groningen, Netherlands.

Bontemps, Y., Heymans, P., Schobbens, P.Y., Trigaux, J.C., 2004. Semantics of feature diagrams, Proceedings of the Workshop on Software Variability Management for Product Derivation (Towards Tool Support).

Bosch, J., 2000. Design and use of software architectures: adopting and evolving a product line approach. Pearson Education (Addison-Wesley and ACM Press), ISBN 0-201-67494-7.

Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, H., Pohl, K., 2001. Variability Issues in Software Product Lines, Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), pp. 11–19.

Beuche, D., Papajewski, H., Schröder-Preikschat W., 2004. Variability management with feature models, Science of Computer Programming Vol. 53(3), pp. 333-352.

Brownsword, L., Clements, P., 1996. A Case Study in Successful Product Line Development, CMU/SEI-96-TR-016, ADA315802.

Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns, SEI Series in Software Engineering, Addison-Wesley, ISBN: 0-201-70332-7.

Coplien, J., Hoffman, D., Weiss, D., 1998. Commonality and Variability in Software Engineering. IEEE Software, 15(6).

Czarnecki, K., Eisenecker, U., 2000. Generative Programming, Addison-Wesley, 2000.

Czarnecki, K., Helsen, S., Eisenecker, U., 2005. Formalizing cardinality-based feature models and their specialization, Software Process Improvement and Practice, 10(1), pp. 7-29.

Dashofy, E. M., Hoek, A. van der, Taylor, R. N., 2001. A Highly-Extensible, XML-Based Architecture Description Language, Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001).

Encoway GmbH website, http://www.encoway.de.

Gomaa, H., Webber, D., 2004. Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model, Proceedings of the Hawaii International Conference on System Sciences, Hawaii.

Griss, M., Favaro, J., d'Alessandro, M., 1998. Integrating Feature Modeling with the RSEB, Proceedings of the Fifth International Conference on Software Reuse, pp. 76-85.

Halmans, G., Pohl, K., 2003. Communicating the variability of a software-product family to customers, Software and Systems Modeling, 2(1), pp.15-36.

Hoek, A. van der, 2004. Design-Time Product Line Architectures for Any-Time Variability, Science of Computer Programming special issue on Software Variability Management, 53(30), pages 285–304.

Hotz, L., Krebs, T., Wolter, K., Nijhuis, J., Deelstra, S., Sinnema, M., MacGregor, J., 2006. Configuration in Industrial Product Families - The ConIPF Methodology, IOS Press, ISBN 1-58603-641-6.

Definition of Inference Engine, http://en.wikipedia.org/wiki/Inference_engine.

Jacobson, I., Griss, M., Jonsson, P., 1997. Software Reuse, Architecture, Process and Organization for Business Success. Addison-Wesley, ISBN: 0-201-92476-5.

Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S., 1990. Feature Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-021.

Keepence, B., Mannion, M., 1999. Using Patterns to Model Variability in Product Families, IEEE Software 16(4), pp. 102-108.

Kiczales, G., Lamping, K., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., Irwin, J., 1997. Aspect-Oriented Programming, Proceedings of ECOOP 1997.

Koala-C Website, http://www.program-transformation.org/Tools/KoalaC.

Krueger, C., 2002. Variation Management for Software Production Lines. In Proc. of the 2nd International Software Product Line Conference, volume 2379 of LNCS, pages 37-48, San Diego, USA, ACM Press. ISBN 3-540-43985-4.

Lichter, H., Maßen, T. von der, Nyßen, A., Weiler, T., 2003. Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung, Technical Report in Aachener Informatik Berichte, ISSN 0935–3232.

Maßen, T. von der, Lichter, H., 2004. RequiLine: A Requirements Engineering Tool for Software Product Lines, 5th Workshop on Product Family Engineering (PFE-5), Springer Verlag Lecture Notes on Computer Science Vol. 3014 (LNCS 3014), pp. 168-180.

Maßen, T. von der, Lichter, H., 2002. Modeling Variability by UML Use Case Diagrams, Modeling Variability by UML Use Case Diagrams, Proceedings of the International Workshop on Requirements Engineering for Product Lines 2002, Technical Report ALR-2002-033.

Medvidovic, N., Taylor, R. N., 2000. A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering, vol. 26, no. 1, pages 70-93.

Myllärniemi, V., Asikainen, T., Männistö, T., Soininen, T., 2005. Kumbang Configurator - A Configuration Tool for Software Product Families, in Proceedings of the Configuration Workshop at IJCAI-05.

Ommering, R. van, Linden, F. van der, Kramer, F., Magee, J., 2000. The Koala Component Model for Consumer Electronics Software. IEEE Computer, p. 78-85.

Pohl, K., Böckle, G., Linden, F. van der, 2005. Software Product Line Engineering: Foundations, Principles, and Techniques, Springer Verlag, ISBN 10-3-540-24372-0.

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

Riebisch, M., Streitferdt, D., Pashov, I., 2004. Modeling Variability for Object-Oriented Product Lines, Workshop Reader of Object-Oriented Technology at ECOOP 2003, Springer, Lecture Notes in Computer Science, Vol. 3013, pp. 165 – 178.

Schmid, K., John, I., 2004. A customizable approach to full lifecycle variability management, Science of Computer Programming, Special issue: Software variability management, Vol. 53, Issue 3, pp. 259 - 284.

Schmid, K., John, I., Kolb, R., Meier, G., 2005. Introducing the PuLSE Approach to an Embedded System Population at Testo AG, Proceedings of the 27th International Conference on Software Engineering (ICSE), pp. 544 – 552.

Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y., 2006. Feature Diagrams: A Survey and A Formal Semantics, Proceedings of the 14th IEEE International Requirements Engineering Conference.

Svahnberg, M., Gurp, J. van, Bosch, J., 2005. A Taxonomy of Variable Realization Techniques, Software Practice & Experience, Vol 35, pp. 1-50.

Trigaux, J.C., Heymans, P., 2003. Modelling Variability Requirements in Software Product Lines: A comparative survey, Technical report PLENTY project, Institut d'Informatique FUNDP, Namur, Belgium.

Weiss, D. M., Lai, C. T. R., 1999. Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley.

World Wide Web Consortium, XML Path Language (Xpath) 2.0, 2005. http://www.w3.org/TR/xpath20.

Zhang, H., Jarzabek, S., 2004. XVCL: a mechanism for handling variants in software product lines, Science of Computer Programming, 53(3), pp. 381-407.

PART II. Modeling

In Part I, we discussed the causes of product derivation issues, and presented a classification that shows the similarities, differences and open issues of variability modeling techniques. These results clearly show that there is enough room for improvement. Based on these results, we therefore developed a collection of techniques called COVAMOF. COVAMOF is a variability management framework that consists of our conceptual understanding of variability, a variability modeling language, a tool-suite, a process, and an assessment technique. This Part presents the modeling concepts and language of COVAMOF.

Chapter 7 Variability Modeling Concepts

In our discussion on the core issues of product derivation, i.e. complexity and implicit properties, we said that there were two ways to address these issues. The first way is by devising solutions that provide a suitable way to effectively deal with these issues. The second way is by addressing the source of these issues, viz. by reducing complexity and implicit properties. In Part I, we already discussed a number of solutions for dealing with complexity in modeling, such as a hierarchical organization of variation points, and the first-class representation of variation points and dependencies. In this chapter, we therefore specifically focus our discussion on the challenges and concepts that are related to different types and availability of knowledge about dependencies, and dependency interaction.

Based on	Section numbers
M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, Modeling Dependencies in Product Families with COVAMOF, Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2006), March 2006.	All sections in this chapter

7.1. Introduction

In Part I, we described how products are derived in software product families (see also Figure 26). Typically, a product derivation process starts with an initial phase, where a first configuration of the product is derived using assembly (construction, or generation), configuration selection, or any combination of both. Although the initial configuration usually provides a sufficient basis for continuing the process, the initial configuration is often not finished. The process then enters the iteration phase, where the configuration is changed until the product is ready.

Product derivation is enabled through variability in the product family artifacts. Variability is the ability of a software system or artifact to be extended, changed, customized, or configured for use in a specific context. On the one hand, variability is enabled through variation points, i.e. the locations in the software that enable choice at different abstraction layers. Each variation point is associated with a number of options to choose from (called variants). On the other hand, the possible configurations are restricted due to dependencies that exist between variants, and the constraints that are imposed upon these dependencies.

In Part I, we determined that the implicit or unknown properties, and the almost unmanageable amount of variability in terms of sheer numbers (ranging up to ten thousands of variation points and dependencies), are two core issues that have a detrimental effect on the time, effort, and cost associated with product derivation. In addition, the implicit and unknown properties make the product derivation process highly dependent on experts.

A large part of the implicit and unknown properties problem is associated with dependencies. During product derivation, engineers specify constraints on the dependencies, such as “the memory usage should be smaller than 64 MB”, or “the compatibility of variants should be TRUE”. These constraints originate from product requirements. In essence, dependencies thus specify a system property whose value is based on the selection of variants at the different variation points. A dependency can, for example, specify a property that tells whether a particular set of selected variants is compatible with each other, or a property that specifies the value of a quality attribute such as performance or memory usage. In our framework, we will therefore group dependencies on the level of variation points.

As we discussed in our classification on variability modeling techniques, these techniques only address the formalization of dependencies. Formalization, however, does not address the most challenging problems in variability management, viz. challenges that result from dealing with imprecise and incomplete knowledge. We discuss these challenges in the sections below, using examples from the Dacolian Case Study presented in Chapter 4.

7.2. Knowledge types: formalized, documented and tacit knowledge

We start our discussion on knowledge types and dependencies with an example of the Intrada product family:

Example 1: *Several parts of the configuration of an Intrada product are automated. This automation is based on the extensive use of formalized knowledge. Examples of formalized knowledge exist at all stages of the configuration process:*

- *At pre-compile time, Dacolian uses #ifdef and #define preprocessor directives to make sure that the platform and operating system dependencies are correctly handled. This means that the appropriate include files, sources, and constants are supplied automatically when the designer selects a platform/operating system combination.*
- *For devices with a limited amount of memory, e.g. PDA’s, Intrada products require different versions of modules to be linked. Dacolian uses specialized tooling to generate the correct Makefiles.*

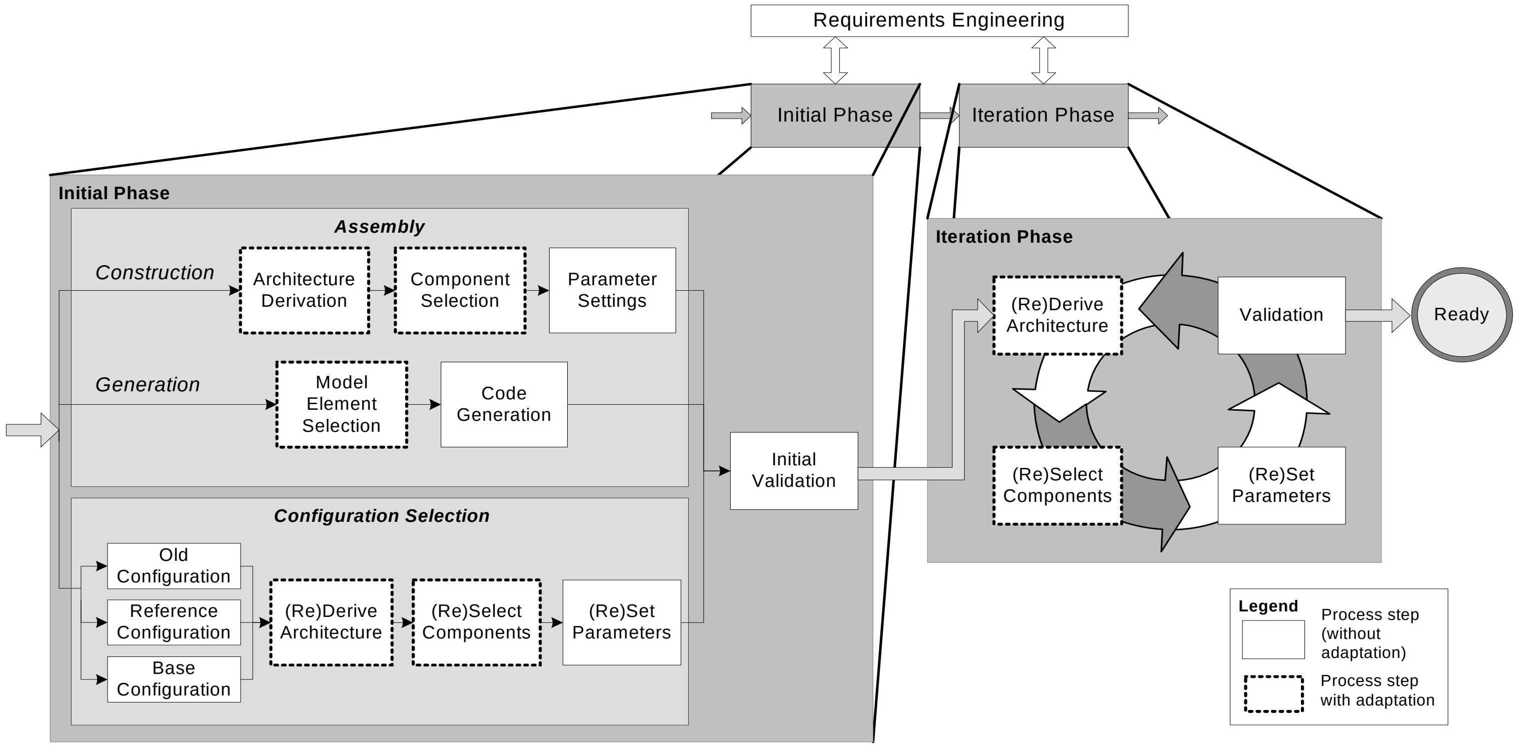


Figure 26 – Product derivation in software product families. A typical product derivation process starts with an initial phase, where a first configuration is constructed using assembly or configuration selection. The process usually goes through a number of iterations before the product is finished.

The knowledge that is available for the dependencies in the example above is an illustration of what we call formal knowledge, i.e. explicit knowledge that is written down in a formal language, and that can be used and interpreted by computer applications. Existing variability modeling approaches are only based on this formalized knowledge (see Part I).

In product families, however, two other types of knowledge exists as well, i.e. tacit knowledge, and documented knowledge. Tacit knowledge (Nonaka and Takeuchi, 1995) is information that exists in the minds of the engineers that are involved in product derivation, but that is not written down in any form. Documented knowledge is information that is expressed in informal models and descriptions. To exemplify both types, we provide two examples below.

Example 2*: The large tolling projects require a high performance from the Intrada products. The derivation of these products is therefore a complicated task. Engineers have established that there are indeed dependencies between the various configuration options, and that tradeoffs have to be made. Graphs like the one in Figure 27 are created for various configurations to score the actual configuration for the dependencies. Dacolian has build tooling to assess a configuration (Intrada Performance Analyser). As the knowledge on how to use this tooling is currently just available as tacit knowledge, only a few experts are capable of performing a directed optimization during the configuration process. Where product derivation for a typical product only takes a few hours at maximum, these complicated product derivations take up to several months.*

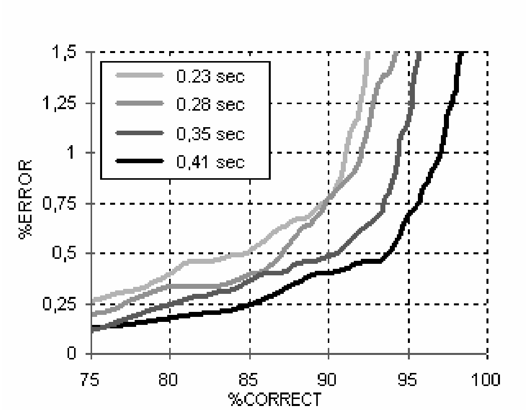
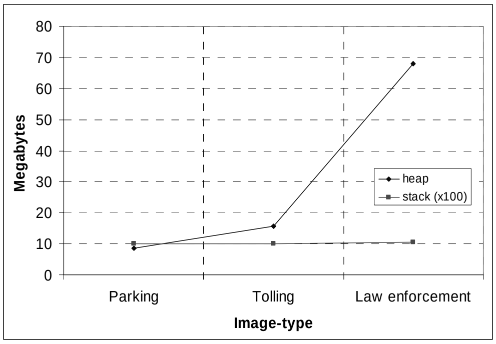


Figure 27 – The effect of varying maximum processing time on a typical configuration. This is an example of partially tacit and partially documented knowledge. Based on such graphs, less experienced application engineers are able to predict the influence of the effect of varying the maximum processing time for recognizing an image. As soon as the maximum processing time becomes a real issue (e.g. max. 0.23 seconds with a correct rate of 95%), only experts know which actions to take in order to perform a directed optimization towards the desired results.

***Example** 3: Dacolian has also documented some of the other complex dependencies, so that less experienced engineers are also capable of deriving a product in a structured way. This example of the Intrada product family involves the dependencies related to memory usage. Different configurations require a different amount of code, heap, and stack memory. Part of the knowledge of how different variants at variation points influence these dependencies has been externalized at Dacolian to documented knowledge (as a series of tables and graphs). The left graph in Figure 3, for example, shows the influence of the selection of different types of matcher variants on the code and data segment. The right graph in Figure 28, on the other hand, shows the influence of a number of reference configurations on the heap and stack memory. These reference configurations consist of the most dominant configuration choices for the dependencies under consideration, according to product type. This knowledge can now be used by a less experienced engineer to determine what choices should be made when memory-size is restricted (e.g. choosing normal matchers, or no matcher at all), or in predicting what the memory usage will be based on matching the product under derivation with the reference configurations.*



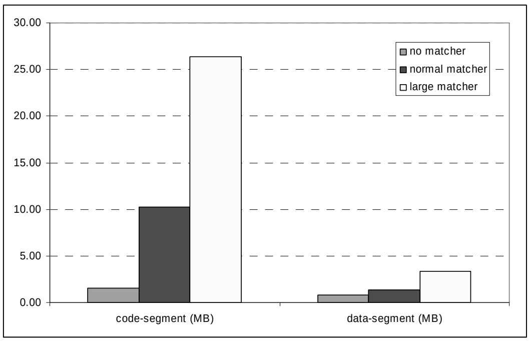


Figure 28 – The influence of the most dominant configuration choices on memory usage. This documented knowledge can now be used to predict the memory usage based on a particular selection, and in directing the product derivation process when the restriction on memory usage is not met.

As the examples above illustrate, tacit and (to a lesser extent) documented knowledge are an integral part of the knowledge that is needed to derive products. They are responsible for the dependency on expert knowledge and manual labor, which is one of the reasons why the product derivation process is a laborious, time-consuming, and error-prone task that requires a number of iterations before a product is finished (see Part I).

There are two ways to address this problem, i.e. making each step faster (in the sense that they take a shorter amount of time, but produce the same result), or improving the result of each of the steps (which will result in less iterations). Fully, or partially automating assembly, configuration selection, and/or validation is a solution that has the potential to do both, but it requires an underlying methodology that enables it.

It may seem logical that, in order to achieve automation during product derivation, we have to transform all tacit and documented knowledge about dependencies into formalized knowledge. While it is true that transforming tacit and documented knowledge into formalized knowledge reduces the dependency on experts and chance of error during product derivation, this assumption runs into a number of issues. First, this so-called process of externalization (Nonaka and Takeuchi, 1995) suffers from the law of diminishing returns (Spillman and Lang, 1924), i.e. that the return on investment in externalization effort decreases as the amount of externalized knowledge increases. Second, tacit knowledge is often described as something that is not easy visible or expressible, and therefore difficult to communicate and share with others (Nonaka and Takeuchi, 1995). Third, depending on the modeling language used, formal specifications can be very hard to maintain, especially if it involves an approach where the only way to specify dependencies are uni- or bidirectional in- or exclusions between variants.

7.3. Imprecise or incomplete knowledge

The existence of different knowledge types is not the only aspect that causes problems, however. When we look at Example 3, for example, the documented knowledge only specifies the average behavior for certain types of choices. The graphs do not specify what the exact memory usage in the data segment is when a specific instance of a particular matcher type is selected. Testing the final configuration may therefore reveal that the actual memory usage is higher or lower than initially expected. A second problem when dealing with dependencies is thus the fact that available knowledge can be imprecise or incomplete as well, which prevents a precise formal specification in the first place. This imprecise and incompleteness can be due to the fact that it is not feasible to put enough effort in the externalization. Often, however, the imprecise and incompleteness is due to the complexity of the situation, as we illustrate below:

Example 4: *All Intrada products are built to interpret outdoor video or still images. It is known that variations in image quality require different Intrada configurations in order to deliver high performance. New projects prove that the available knowledge is both imprecise and incomplete. For an image resolution of 2.7 pixels/cm, for example, Dacolian expected that an image recognition rate of 95% correct, with 0.1% error should be feasible. Later, however, It was discovered that there are complex relations between variation points that are controlled or influenced by aspects such as Signal to Noise ratio, weather conditions, contrast, and country modules used.*

Despite all the effort they put in, Dacolian has not been able to describe all, for configuration relevant, relations between image characteristics and recognition and error rates in a formal or documented way. They currently have to build the system and test it under these conditions to verify the desired value. Their best hope so far is to characterize the image quality into categories that are typical for a certain application area. Dacolian describes these categories by a set of typical images.

Imprecise and incomplete knowledge is typically a problem when the estimated value for a dependency approaches the constraint imposed on the dependency for a product. For example, software engineers can usually be more confident that a constraint of a minimal 80% correct rate (see Figure 27) will be met when the *estimated* correct rate is 90%, than when the estimated correct rate is 81%.

Combined with variation points that influence the value of multiple dependencies, imprecise and incomplete knowledge is a second reason for the necessity of iterations. When, during testing, it proves that certain constraints are not met, the reselection of variants can have an unpredictable effect on multiple dependencies. This can result in multiple trail-and-error iterations. In addition, when it turns out these iterations will not have the apparent positive result, the dependencies have to be weighed against each other (e.g. the correct and error rates in Example 2).

7.4. Dependency Interaction

The variability in industrial product families typically contains a large number of dependencies, each one associated with a number of variation points ranging from one to almost all variation points. Therefore, a lot of variation points are typically associated to more than one dependency. As a result, configuring a variation point in the context of the optimization of one dependency may influence the system property value of all other dependencies that share that variation point. We refer to this mutual influence between dependencies as *dependency interaction*. As the optimal values of those dependencies are often contradicting, such as for the dependencies "CorrectRate" (Example 2), "StackSize" (Example 3), the optimization of dependencies is a complicated task and involves an extensive amount expert knowledge.

7.5. Conclusion

In the classification of variability modeling techniques in Chapter 6, we showed that existing techniques only focus on supporting formal dependencies. However, we only briefly discussed why supporting formal dependencies alone is not enough. In this section, we elaborated on that discussion. We discussed and exemplified several concepts related to dependencies (knowledge types, the nature of imprecision and incompleteness, and dependency interaction) and showed how these concepts are related to iterations in the product derivation process.

Due to the impact on the derivation process, a variability modeling language that adequately supports these concepts is an important step in realizing the goal of reducing expert involvement and number of iterations (see Chapter 5). Since the existing variabilty modeling techniques do not sufficiently deal with these aspects, we developed our own variability management framework COVAMOF. We present this framework in the remaining parts of this thesis. We start by presenting the COVAMOF language and tool-suite in the next chapter.

7.6. References

Dacolian website, http://www.dacolian.com

Nonaka, I., Takeuchi, H., 1995. The Knowledge-Creating Company: How Japanese companies create the dynasties of innovation, New York: Oxford University Press.

Spillman, W.J., Lang, E. 1924. The Law of Diminishing Returns.

Chapter 8 The COVAMOF Modeling Language and Tool-Suite

The first tangible parts of our framework are the COVAMOF Modeling Language and its practical realization in the COVAMOF-VS Tool-Suite (see Figure 29). They capture our conceptual understanding of how variability manifests itself, as well as how variability should be modeled to address the modeling requirements, i.e. effectively handling the imprecise, tacit and documented knowledge, dependency interaction, and the large number of variation points, variants, and complex dependencies. In this chapter, we present the main concepts and modeling elements of COVAMOF. We illustrate the modeling elements with examples from the Dacolian Case Study (see Chapter 4).

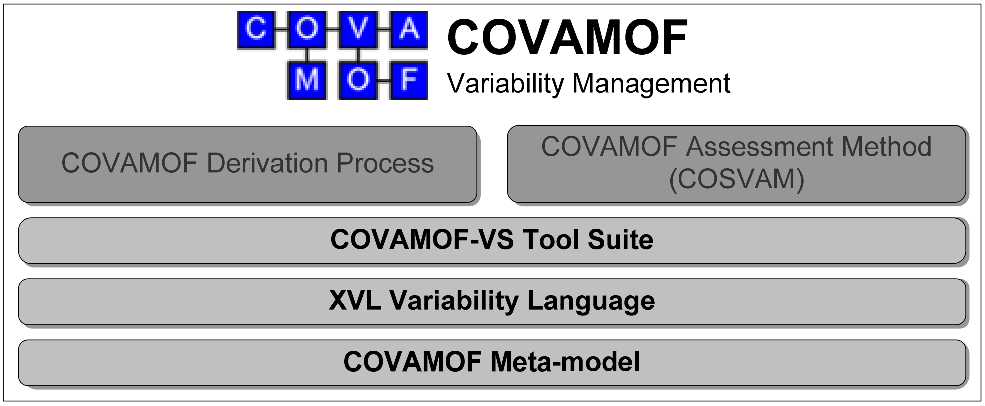


Figure 29. COVAMOF. COVAMOF is a variability management framework that is currently built-up from five main parts, including COSVAM.

Based on	**Section numbers**
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.	Examples in Section 8.2
M. Sinnema, S. Deelstra, Industrial Validation of COVAMOF, Elsevier Journal of Systems and Software, Vol 81/4, pp. 584-600, 2007.	Section 8.1 and 8.3
M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, Modeling Dependencies in Product Families with COVAMOF, Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2006), March 2006.	Section 8.2
M. Sinnema, S. Deelstra, and P. Hoekstra, The COVAMOF Derivation Process, Proceedings of the 9th International Conference on Software Reuse (ICSR 2006), Springer-Verlag Lecture Notes in Computer Science Vol. 4039 (LNCS 4039), pp. 101-114, June 2006.	Screenshot in Section 8.3

8.1. Introduction

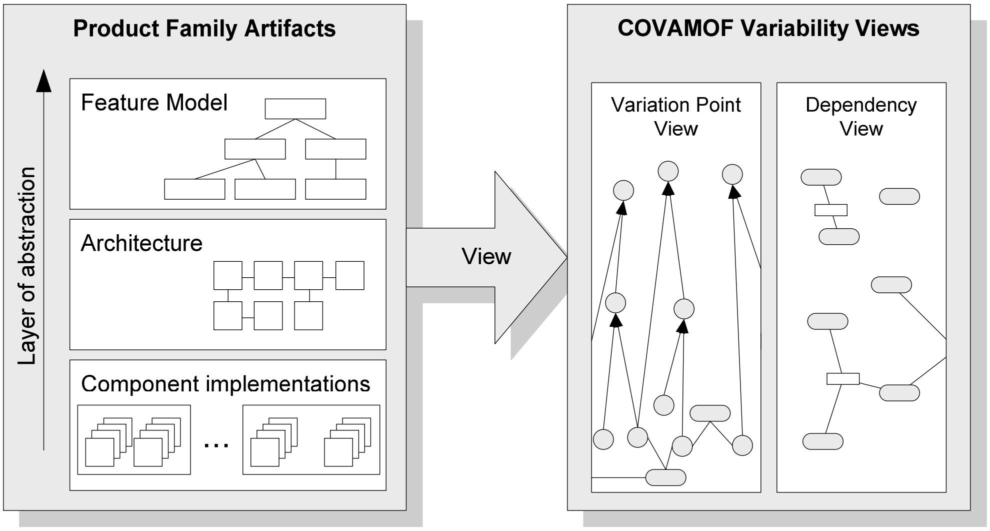


Figure 30. COVAMOF Variability Views. Variation Points and Dependencies are first-class entities that enable different views on the variability in the product family artifacts.

COVAMOF is a variability management framework we developed in order to deal with the issues of expert dependency and large number of iterations. It is a framework where variation points and dependencies are modeled uniformly over different abstraction levels (e.g. features, architecture and implementation), and as first-class citizens. The framework enables different views on the variability provided by product family artifacts (called COVAMOF Variability Views or CVV), such as the variation point view, and the dependency view (see Figure 30).

8.2. The Language

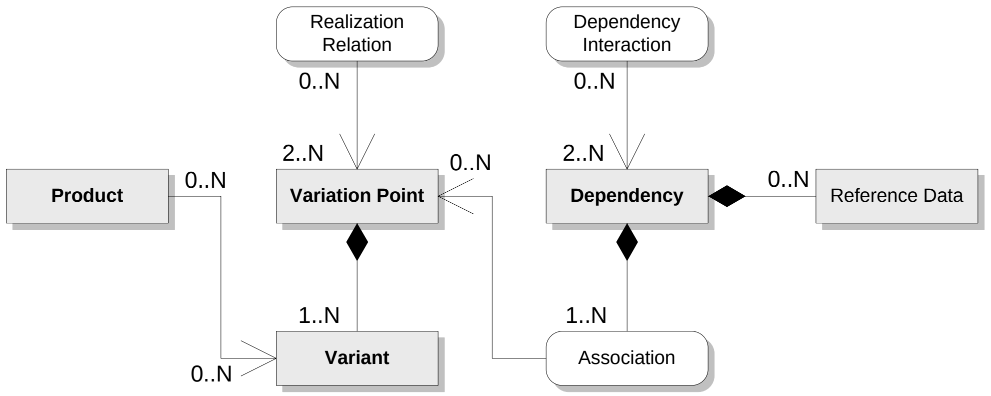


Figure 31. The COVAMOF Meta-model.

The COVAMOF Variability Views capture 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 32. We describe the entities, and relations between them, in more detail below.

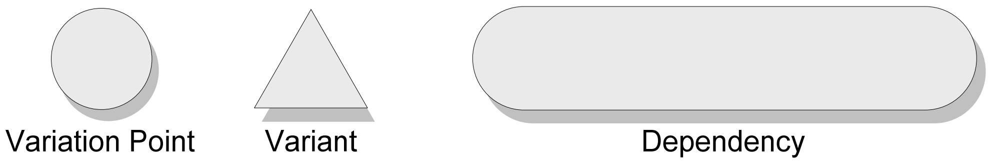


Figure 32. Graphical notation in the COVAMOF Variability Views.

8.2.1. Variation Points and Variants

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 a Variant or value. The Variants and values refer to artifacts in the product family, such as features, architectural components, C header files, or parameters. The COVAMOF language defines five types of Variation Points in the CVV, i.e. optional, alternative, optional variant, variant, and value. Figure 33 presents the graphical representation of these types.

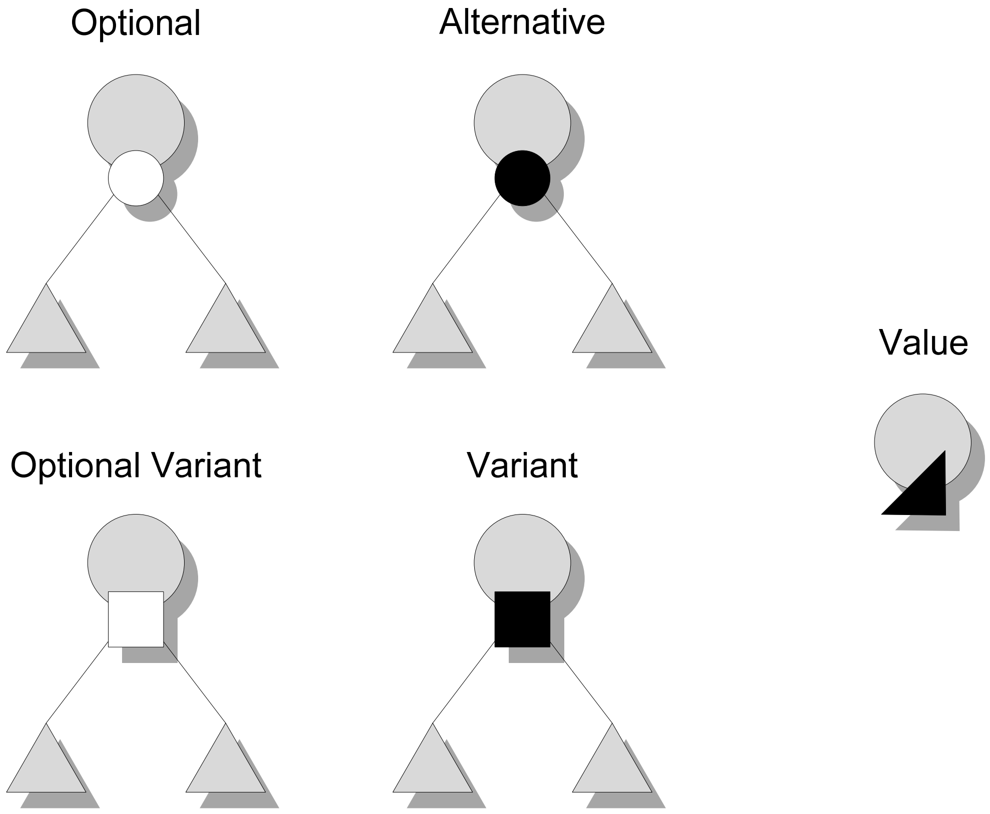


Figure 33. The graphical notation of the five types of variation points in the CVV.

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, 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. With the *binding* of a Variation Points we refer to the specific selection of Variants or values for that Variation Point in a specific product.

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.

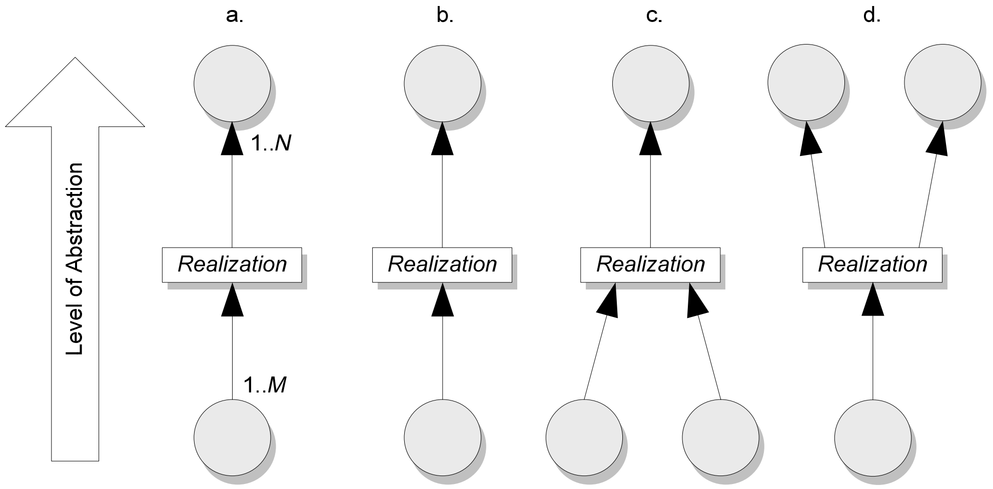


Figure 34. The possible number of variation points associated with a realization relation (a.), a variation point realizing one other variation point on a higher level of abstraction (b.), two variation points realizing one variation point (c.), and one variation point realizing two variation points (d.).

8.2.2. 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 34a). The Realization Relation in the CVV contains rules that describe how a configuration of Variation Points directly depends on the configuration of the Variation Points in a higher level of abstraction in the product family. Figure 34 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.

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

Example 5: *Figure 35 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 chunks of memory that contain the image data. VP Image Source shows the available variants of which one should be selected as image source. This variation point is realized by variation point VP Image Intake, which offers two variants that accommodate for the alternatives of VP Image Source.*

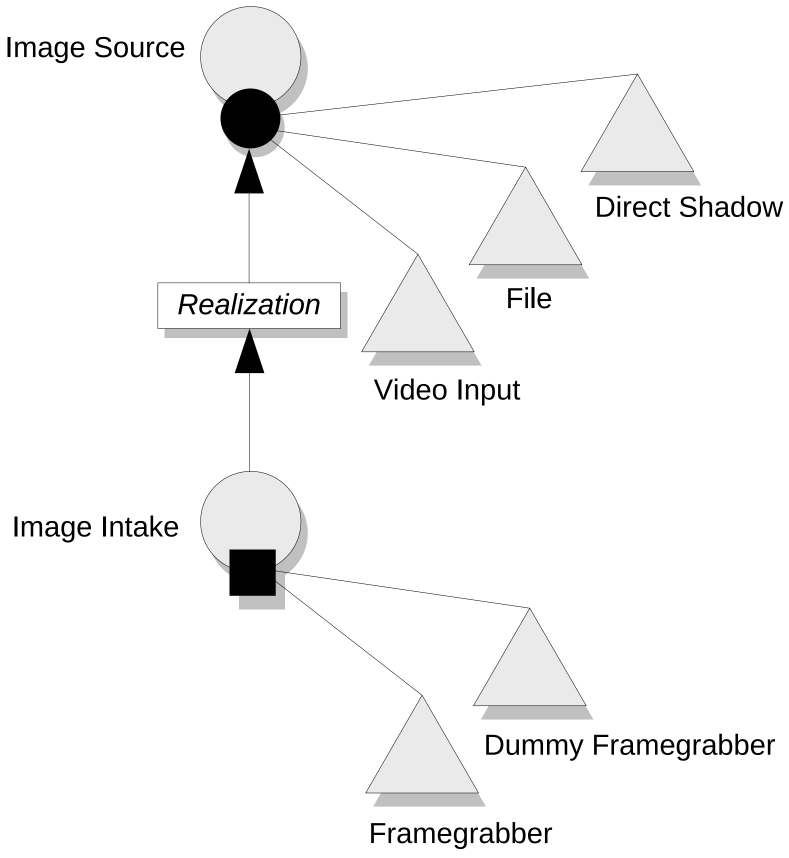


Figure 35. Realization example. One variation point realizing one other variation point at a higher level of abstraction

Example 6: *Figure 36 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 VP Countries, where each variant represents one country. At a lower level, VP Countries is realized by three variation points, VP Neural Network, VP Syntax, and VP Matcher. Which variants of these three variation points are included in the derived products depends on their logical "requires"-dependencies with VP Countries. VP Neural Network is an optional variant variation point and its variants are only used when there is no speed limitation to improve the performance.*

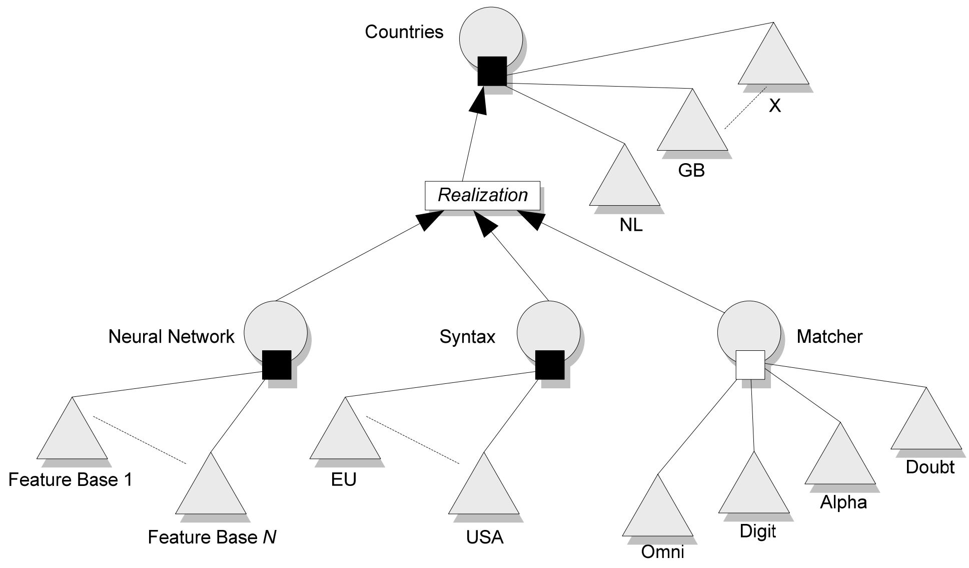


Figure 36. Realization example. Example of multiple variation points realizing one variation point

Example 7: *Figure 37 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 (VP Access Rights). Also various methods of transaction logging, statistic, and error reporting can be selected (VP Reports). Both VP Access Right and VP Reports are realized by VP Database which provides database and search facilities.*

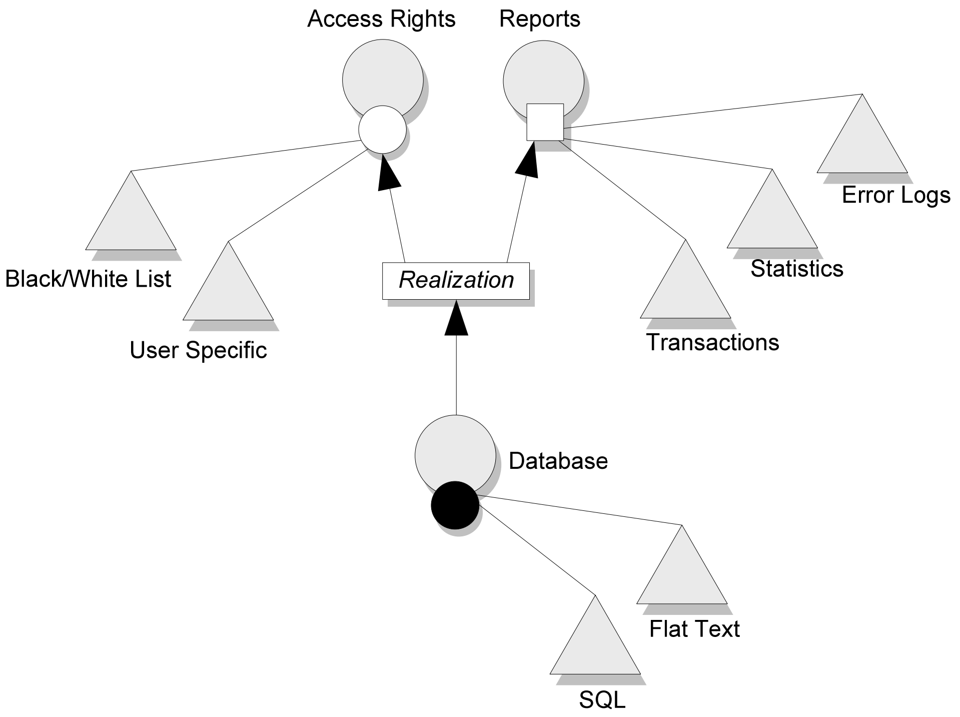


Figure 37 – Realization example. Example of one variation point realizing two variation points at a higher level

8.2.3. Dependencies

Analogous to the description of dependencies in Chapter 7, Dependencies represent a system property and specify how the binding of the Variation Points influences the value of the system property. An example of a system property is a formal dependency that maps pre-compiler directives to {true, false}. Other examples of system properties can be found in Chapter 7 (see also the text in the example below).

***Example 8**: Example 2 and Example 3 in Chapter 7 map a configuration of variation points onto an error rate between 0% and,100%, and onto a required stack between 0MB and, 1024MB, respectively. Figure 38 shows how we visually represent these two Dependencies and system properties in COVAMOF.*

ErrorRate
[0-100]%
StackSize
[0-1024]MB

Figure 38. Dependency examples. Two examples of Dependencies from the Dacolian case study.

As shown in Figure 39, Dependency entities contain six attributes, i.e. the SystemPropertyName, SystemPropertyConstraint, the Associations, the SystemPropertyFunction, the DocumentedKnowledge and the ReferenceData. Below, we explain how the mapping from a binding of the Variation Points to the Dependency values is captured in the variability model, by showing the purpose of each of these attributes.

- **SystemPropertyName:** This attribute contains a string representing the name of the system property, e.g. “ErrorRate” or “RequiredStack”.
- **SystemPropertyConstraint:** The Dependency entities furthermore specify a valid range of the system property. As this range may vary between product instances, the constraint may contain parameters. An example of a product constraint from Example 2 is “ErrorRate ≤ [MAXErrorRate]”. In this case, for each product the MAXErrorRate has to be specified and the actual value of the ErrorRate system property should not exceed MAXErrorRate.
- **Associations:** We refer to the set of Variation Points that influence the system property of the Dependency as the *associated variation points*. For each associated Variation Point, an Association instance is contained in the Associations attribute. The Association class is described in the following section.
- **SystemPropertyFunction:** This attribute contains a (*partial*) function from the binding of the associated Variation Points to a (estimated) value for the system property. This function is specified formally and is constituted from the formal knowledge that is available on the mapping. For example, the code segment usage of Example 3 can be exactly calculated from the matcher Variants that have been selected (see also Figure 36). An estimated value can be refined by the information in the Association entities (see Section 8.2.4).
- **DocumentedKnowledge:** This attribute is a set of references to documented knowledge. The contents of a reference can range from a URL, e.g. to a MS Excel file, to the contact information of a product family expert. The engineers can use documented knowledge during the product derivation process to obtain a reasonable first guess of the binding of the Variation Points. Furthermore,

they can use it to estimate the value of a system property based on the binding of the associated Variation Points. Figure 27, for example, illustrates documented knowledge from the Intrada product family. It contains a graph with data about the system property "ErrorRate" and "CorrectRate", determined by testing the effect of several maximum processing times.

- **ReferenceData:** This attribute contains a set of ReferenceDataElement entities. The purpose and properties of these entities are described in Section 8.2.2.

The model combines two ways of capturing the knowledge about how the configuration of the associated variation points maps to a value in the target domain, i.e. by Association entities and ReferenceData entities.

8.2.4. Associations

For each associated Variation Point, an Association entity is contained in the Dependency entity. The VariationPoint attribute of an Association refers to a Variation Point entity in the variability model (see Section 8.2.1). COVAMOF distinguishes three different types of Associations, i.e. abstract, directional and logical Associations. The type of an Association is stored in the AssociationClass attribute of an Association entity. These types relate to the type and completeness of the knowledge we discussed in Chapter 7:

Abstract: On the first level, Dependencies contain abstractly associated Variation Points. The only information product family experts have on abstractly associated Variation Points is that its configuration influences the dependency values of the Dependency. There is no information available on *how* a (re)binding will effect these values.

Directional: On the second level, Dependencies contain directionally associated Variation Points. These Associations do not only specify that its configuration influences the values of the Dependency, but it also describes some information on *how* the value depends on its binding. The effect of a (re)binding of a directionally associated Variation Point is not necessarily fully known and documented, and is not specified in a formal manner. The Association entity, however, describes this effect in the Impact attribute as far as it is known to the experts. This information can be used to refine an estimated value from the SystemPropertyFunction of the Dependency.

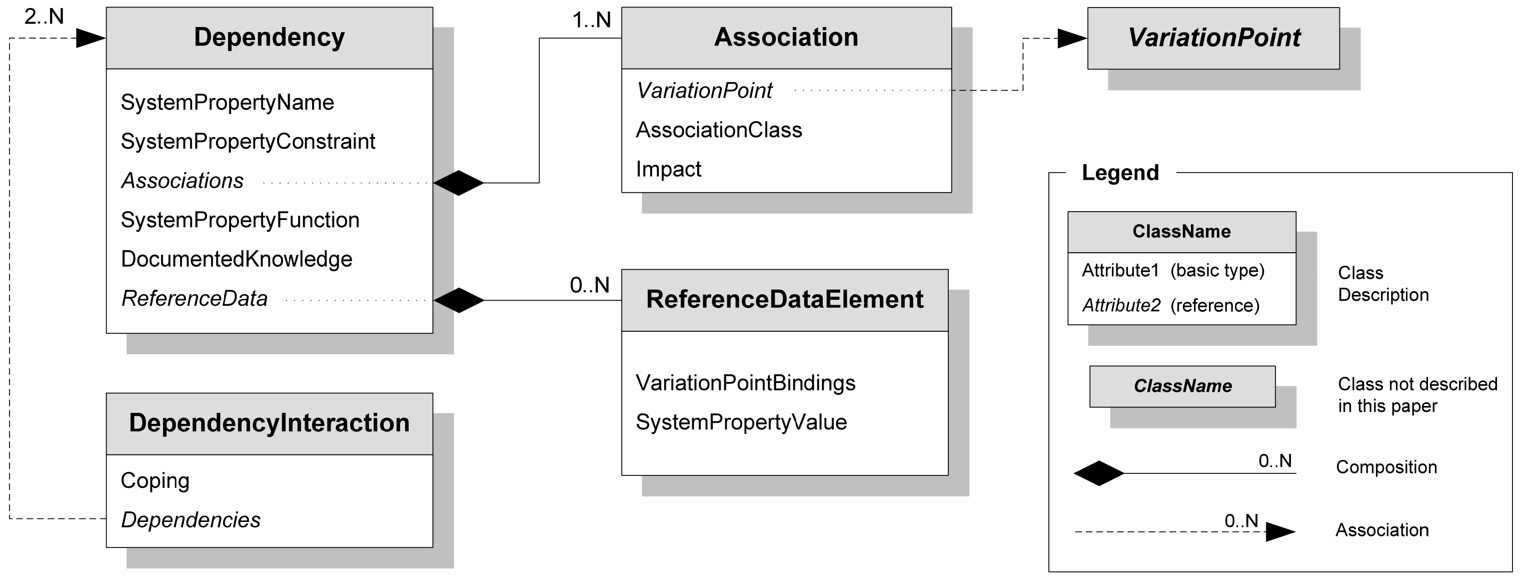


Figure 39. Dependencies in the COVAMOF Meta-model. Dependency entities contain one or more Associations to variation points and zero or more Reference Data elements. The DependencyInteraction entities specify relations between two or more Dependency elements.

Logical: On the highest level, Dependencies contain logically associated Variation Points. The effect on the Dependency values of the Dependency is fully known and is specified as formal knowledge. This specification is integrated in the SystemPropertyFunction attribute of the Dependency entity.

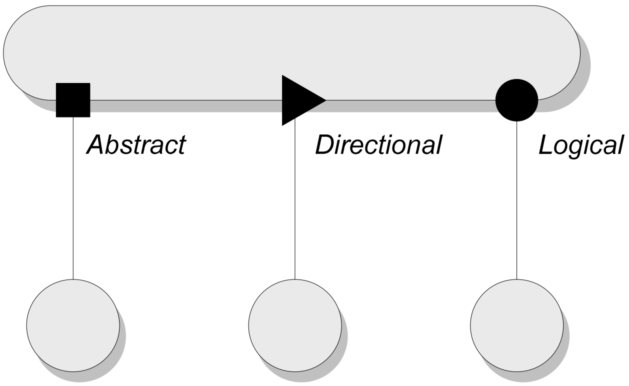


Figure 40. Graphical representation of a dependency and its associations.

The type of associations of a Dependency influences the type of the Dependency. In COVAMOF, we distinguish between Statically Analyzable and Dynamically Analyzable Dependencies. For Statically Analyzable Dependencies, the exact value can be calculated before running the system, which requires Logical Associations. The exact value of Dynamically Analyzable Dependencies can only be determined by running and measuring the system, which is the case if the Dependency contains Abstract and Directional Associations.

Example 9 *The system properties of Example 3 in Chapter 7, i.e. "CodeSegmentSize", "StackSize", and "HeapSize", illustrate all three Association types. The total memory required for a product consists of the size of the code segment of Intrada, the maximum stack size and the maximum heap size. The memory required by the code segment is exactly known and calculated upfront from the component selection. All Variation Points related to the component selection are therefore logically associated in the variability model. For the stack-size, however, it is not precisely known, but software engineers can predict beforehand whether a rebinding of Variation Points related to the stack size will result in an increase or decrease of required stack space. Therefore, these Variation Points are directionally associated in the variability model. The effect of the binding of Variation Points on the maximum heap size is largely unknown and can only be determined by testing configurations. These Variation Points are therefore abstractly associated to the "HeapSize" Dependency.*

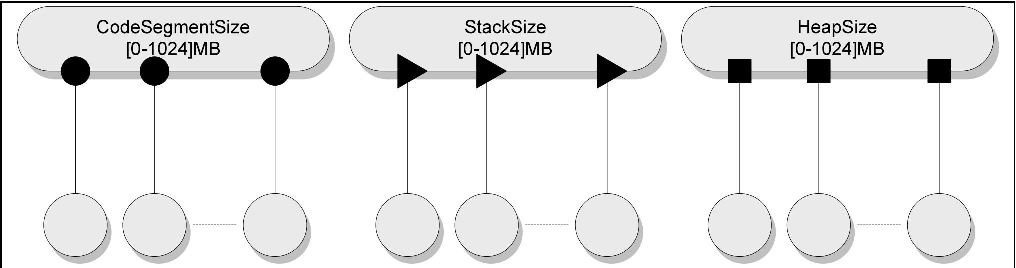


Figure 41. Graphical representation of the dependencies related to memory usage.

8.2.5. Reference Data Element

In addition to the SystemPropertyFunction and the Associations, Dependencies contain reference data about the mapping from the binding of associated Variation Points to the value of the system property. This reference data is collected during testing, and is defined by a set of ReferenceDataElement entities. Each Reference Data Element entity contains a binding of variation points (VariationPointBindings attribute) together with the value of the system property for that specific binding (SystemPropertyValue attribute).

Similar to the DocumentedKnowledge attribute of Dependency entities, reference data can be used in two ways. On the one hand, it can be used to find starting points for the derivation process. A Reference Data Element can be selected whose specific SystemPropertyValue is closest to the required value for the product being derived. The bindings of this reference data element can be used as a starting point. On the other hand, the reference data can be used during the derivation process to estimate the value of a system property based on a binding of Variation Points.

8.2.6. Dependency Interaction

In Chapter 7, we described the concept of dependency interactions. Although the sets of dependencies that interact can easily be generated from the Dependency entities and their Associations, the COVAMOF variability model also explicitly captures Dependency Interaction entities in the variability model. The Coping attribute of Dependency Interaction entities specify, for a set of Dependencies, how to cope with the shared associated Variation Points during product derivation. This textual specification is documented by product family experts and contains a strategy for developing a reasonable first guess during the initial phase, a strategy to optimize the values in the iteration phase, as well as guidance for making trade-offs between interacting Dependencies.

Example 10 *In order to illustrate the dependency interactions in the CVV, we present an example from the Dacolian case study in Figure 42. There is a mutual interaction between the two dynamically analyzable dependencies "StackSize" and "ErrorRate". Both dependencies include the variation point "Matcher" (see also Figure 36). Including additional matcher variants will improve the "ErrorRate" dependency but downgrade the "StackSize" 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.*

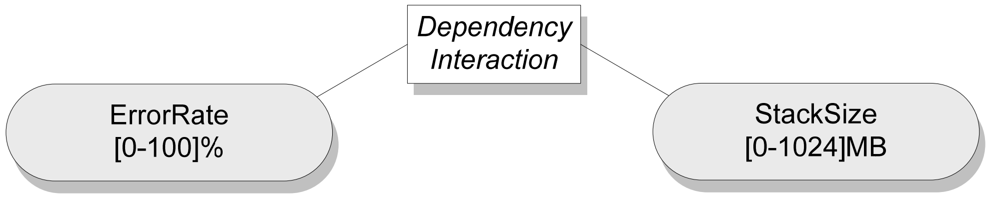


Figure 42. An example of dependency interaction in the Intrada product family.

8.2.7. Products

Product entities specify the Variants and values that have been selected for a product configuration, as well as the restrictions that are posed on the Dependency values. The Product entities furthermore contain information about the customer of the product.

8.3. The Tool-Suite

COVAMOF is supported by a tool-suite, called COVAMOF-VS (see Figure 43). This tool-suite is implemented as a combination of Add-Ins for Microsoft Visual Studio (Microsoft Visual Studio website) and provides an integrated variability view on the active Solution, which contains the product family artifacts. Examples of those artifacts are XML-based feature models, start-up parameter settings and C++/C# source files. The variability information in these artifacts is either directly interpreted from language constructs (such as #ifdef), or from special constructs of the COVAMOF variability language XVL. While we eventually strife for extending a programming language with these constructs, variability information in code files is currently primarily inserted as comments that follow the XVL syntax.

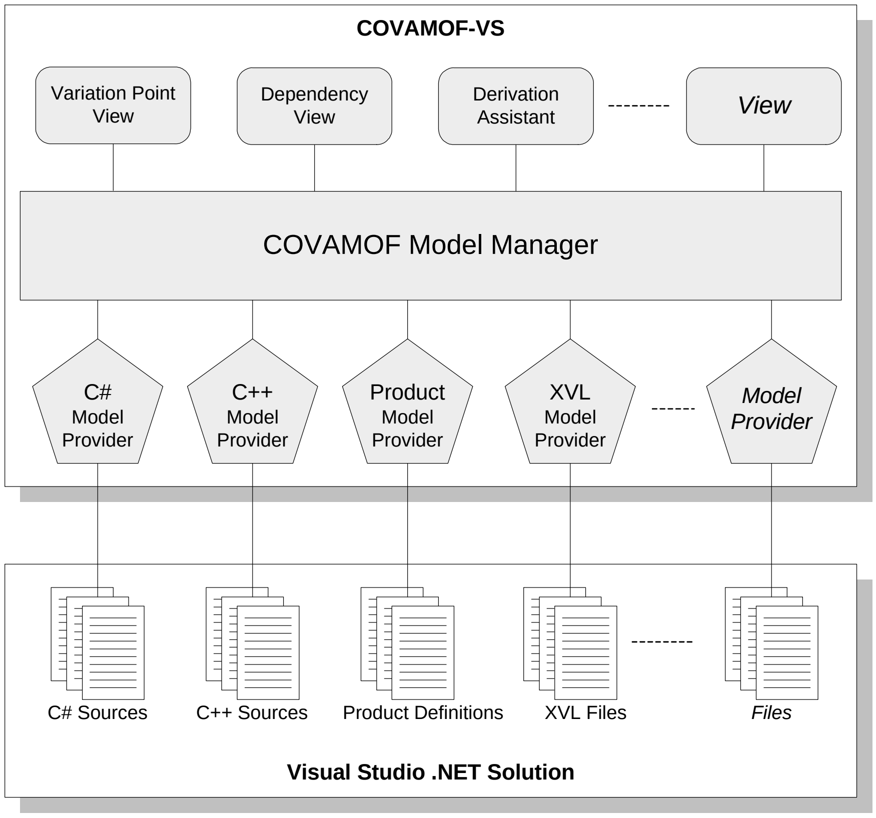
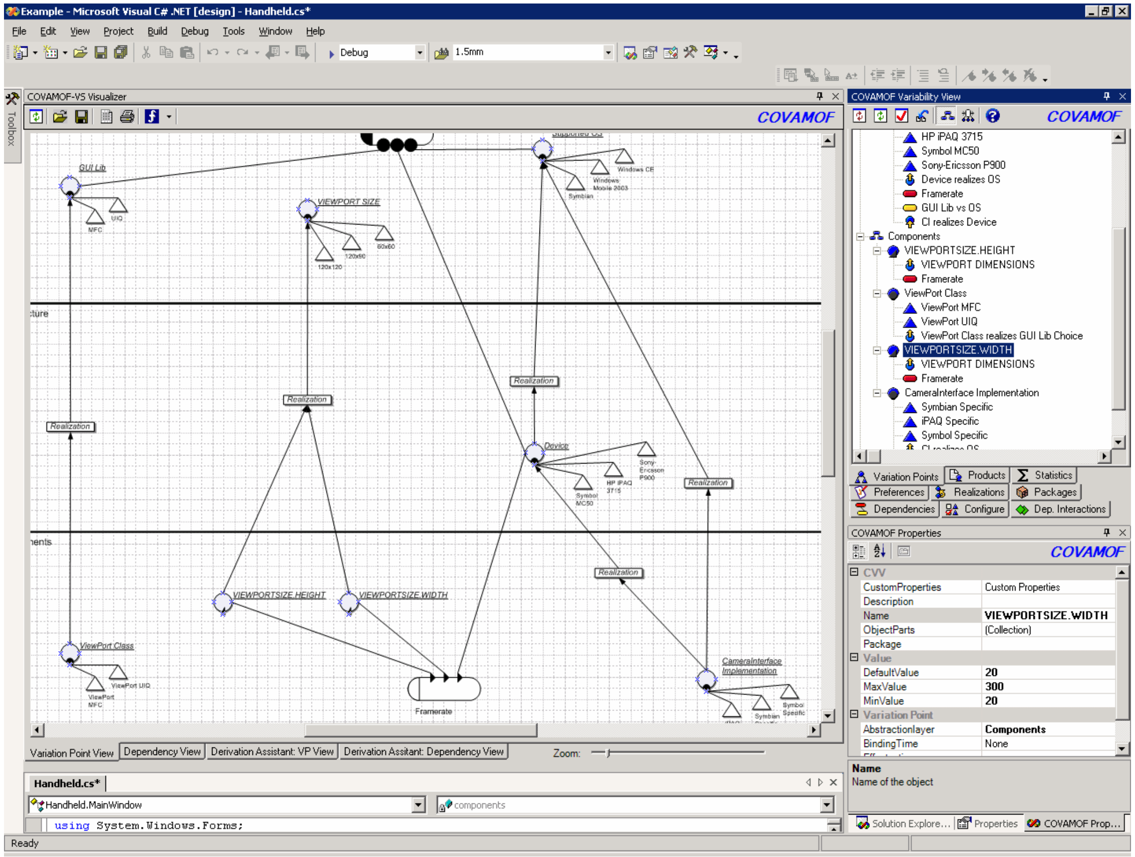


Figure 43. COVAMOF-VS. The COVAMOF Visual Studio Add-in maintains a variability model that is constructed from Solution files through pluggable model providers, and that can be visualized and configured through the pluggable View components.

The extraction of variability information is done by plug-in components that register themselves on one or more file types. The components are called Model Providers. Model Providers convert the extracted variability to entities in the integrated variability model of COVAMOF. They are also responsible for feeding additions and changes to the variability in the Views directly back into the files of the MS Visual Studio Solution. A screenshot of the tool is provided in Figure 44.



HP iPAQ 3715
Symbol MC50
Sony-Ericsson P900
Device realizes OS
Framerate
GUI Lib vs OS
CI realizes Device
Components
VIEWPORTSIZE.HEIGHT
VIEWPORT DIMENSIONS
Framerate
ViewPort Class
ViewPort MFC
ViewPort UIQ
ViewPort Class realizes GUI Lib Choice
VIEWPORTSIZE.WIDTH
VIEWPORT DIMENSIONS
Framerate
CameraInterface Implementation
Symbian Specific
iPAQ Specific
Symbol Specific

CVV	
CustomProperties	Custom Properties
Description	
Name	VIEWPORTSIZE.WIDTH
ObjectParts	(Collection)
Package	
Value	
DefaultValue	20
MaxValue	300
MinValue	20
Variation Point	
Abstractionlayer	Components
BindingTime	None

Name
Name of the object

Handheld.cs*
Handheld.MainWindow
components

```
using System.Windows.Forms;
```

Figure 44. A screenshot of COVAMOF-VS. This figure shows how variation points and dependencies are visualized by the COVAMOF tool-suite.

PART III. Derivation

Based on the concepts and modeling language of Part II, we developed a product derivation process. This process is an instance of the generic product derivation process we discussed in Part I, and is called the COVAMOF Derivation Process. It is specifically tuned to the elements of our framework. In this Part we present this process and its validation.

Chapter 9 The COVAMOF Derivation Process

As a practical realization of COVAMOF (see Figure 45), we developed the COVAMOF-VS tool suite, which provides several variability views on C#, C++, Java, and many other types of projects in Microsoft Visual Studio .NET. One of its purposes is to support the application engineers during product derivation. In this chapter, we show how the combination of COVAMOF and COVAMOF-VS facilitates an engineer during product derivation, and what benefits are gained by it.

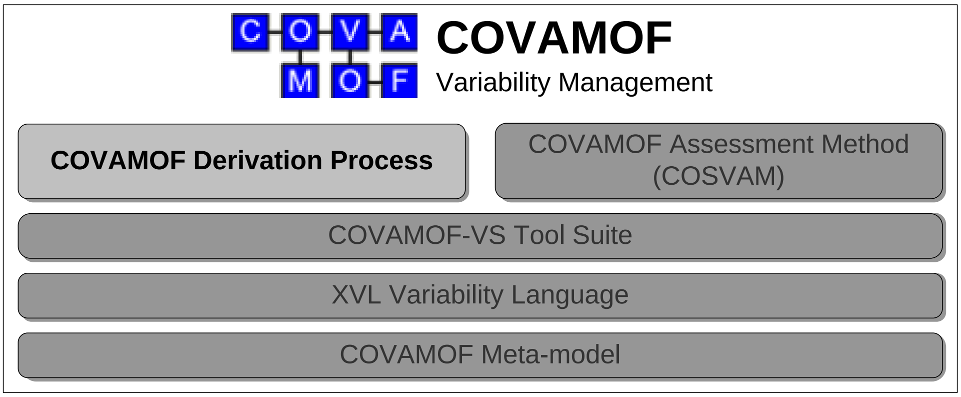


Figure 45. COVAMOF. COVAMOF is a variability management framework that is currently built-up from five main parts, including the COVAMOF Derivation Process.

Based on	Section numbers
M. Sinnema, S. Deelstra, and P. Hoekstra, The COVAMOF Derivation Process, Proceedings of the 9th International Conference on Software Reuse (ICSR 2006), Springer-Verlag Lecture Notes in Computer Science Vol. 4039 (LNCS 4039), pp. 101-114, June 2006.	Section 9.1 and Section 9.2. Note that we left out the sections that covered the COVAMOF model, since this was already discussed in Chapter 8.
M. Sinnema, S. Deelstra, Classifying Variability Modeling Techniques, Elsevier Journal on Information and Software Technology, Vol. 49, pp. 717–739, July 2007.	Section 9.3

9.1. The Process

Product derivation is the construction of a software product that is built by selecting and configuring product family artifacts (see Chapter 3). With COVAMOF, products are derived by the COVAMOF Derivation Process, which as in instantiation of the generic derivation process we presented in Chapter 3. The COVAMOF Derivation Process allows organizations to gain maximum benefit from COVAMOF and its associated tool support. It is divided into four steps, i.e. Product Definition, Product Configuration, Product Realization, and Product Testing. As visualized by Figure 46, the last three steps can occur in one or more iterations. The following subsections describe each of these four steps.

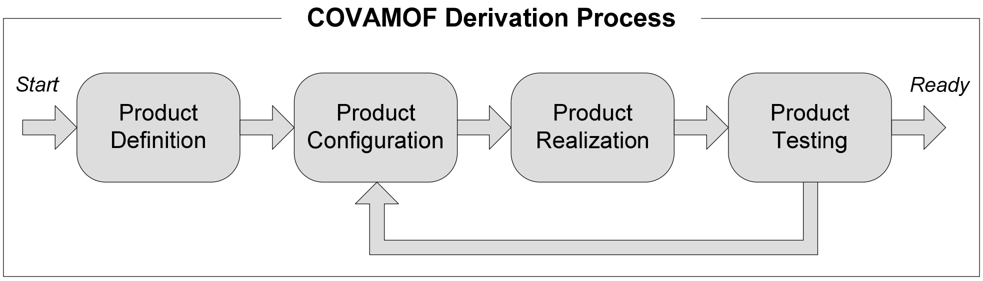


Figure 46. The COVAMOF Derivation Process. This iterative process breaks down into four steps, i.e. Product Definition, Product Configuration, Product Realization and Product Testing. It is an instance of the generic derivation process we discussed in Chapter 3.

9.1.1. Product Definition

The first step of the COVAMOF Derivation Process is called Product Definition. In this step, the engineer creates a new Product entity in the variability model. In response, COVAMOF-VS stores this product in the active Solution of Visual Studio. The properties of Product entities are the customer, a unique name for the product, and variation points that have been bound for the product. The latter are described by bindings of variation points, i.e. combinations of variation points together with their selected variants or parameter value. The engineer can directly fill in the customer and name property. For the bindings of the variation points, one or more iterations of the Product Configuration step are required.

9.1.2. Product Configuration

In order to start the Product Configuration step, the engineer selects the Product entity from the available products dropdown menu in the COVAMOF-VS toolbar. From that point, COVAMOF-VS is in configure mode and additional configuration information about the product at hand is shown in both variability views.

When the variation point view of COVAMOF-VS is in configure mode, the variation points that are bound for the product at hand are marked by a different color. During the Product Configuration step, the engineer binds one or more variation points to new values or variants based on the customer requirements. In order to bind these variation points, he marks the new variants or specifies the new values in the variation point view. The order in which variation points are bound is dynamically determined by the realization relations and dependencies in the variability model.

Meanwhile, the binding of a variation point is effectuated by creating or updating the relation between the Product entity and the Variation Point entity. Each variation point that is bound triggers the inference engine and the validation engine to work:

Inference Engine: As described in Chapter 8, the rules of Realization Relations define how variation points on a lower level of abstraction realize variation points on a higher level of abstraction. These rules are used by the inference engine, to automatically bind variation points on a lower level of abstraction. The binding of these variation points is based on the binding of the variation points on a higher level of abstraction. For each rule that the variation point is involved in, the consequences are recursively determined.

Validation Engine: After a variation point is bound by the engineer and the inference engine has worked, the validation engine automatically checks whether no dependencies have been violated. The rule of each statically analyzable dependency is checked and for each dynamically analyzable dependency the reference data elements are checked. Based on the rules, the associations and the reference data, the dependencies in the variation point view are provided with the new estimated value. Any violations of these values with respect to the required values are immediately fed back to the engineer by marking the dependencies in the variation point view with the color red.

The customer requirements can be separated into functional requirements and non-functional requirements. Therefore, the engineer basically has two main concerns during product configuration. First, the right features and components should be selected so that the functional requirements are met. Second, the configuration is tuned and adapted to meet the non-functional requirements. The Realization Relation and Dependency entities in the variability model support the engineer in configuring a product that meets the functional as well as the non-functional requirements. *How* these two entities can be used is explained below. Note that in practice, the engineer has to take both into consideration at the same time.

Realization relations: In order to select the right features and components, the engineer binds variation points in the feature layer. The variation points in the architecture layer can be bound using rules of the Realization Relations between the feature layer and the architecture layer. In case the inference engine was unable to automatically bind these variation points in lower layer of abstraction, the engineer has to bind them manually. Similarly, the variation points in the component layer can be bound by using the realization relations between the component layer and the architecture and feature layer.

Dependencies: The Dependency entities in the variability model support the engineer in binding variation points in such a way that the product is consistent and meets the non-functional requirements. When Product Configuration starts, the engineer specifies, for each of the dependencies the specific value or range that is required for the product at hand. The variation point view of COVAMOF-VS in configure mode shows the (estimated) value of each of the dependencies together with the required value or range. When the actual value is outside the required range, i.e. when the Dependency is violated, the dependency in the variability view is colored red.

The goal of engineers is to (re)bind the variation points in such a way that none of the dependencies are violated. During iterations, the engineer therefore has to shift the value of each violated Dependency into the required range. The information in the Associations and the Reference Data of the Dependency entity are used to determine *how* the current value can be changed in order to meet the required range.

As Logical Associations specify exactly how the (re)binding of a variation point changes the value, this formalized knowledge can easily be used to change the value of the Dependency, and the effect is immediately visible in the variation point view. The documented knowledge in the Directional Associations can be used to increase or decrease the value in the right direction. However, a Product Test (section 9.1.4) is required to determine the new value of the Dependency. How Abstract Associations can be used can only be determined by any reference data available. Otherwise, the engineer has to use the tacit knowledge of an expert or even trial-and-error to change the value of the Dependency.

Note that the process of changing the values of Dependencies is a very complicated task. This is particularly due to the interaction between Dependencies like we described in Chapter 7 and 8. In such situations, the engineer uses the information of the corresponding Dependency Interaction entities in the dependency view (see also Part II). This information helps the engineer in making reasonable trade-offs between system properties and a strategy to accomplish the acceptable values, for example, which Dependency should be resolved first and which should be resolved later in the process.

Usually, the focus during the initial iteration of the COVAMOF Derivation Process is on satisfying the functional requirements, and, as the product functionality becomes more and more fixed, the focus gradually shifts to satisfying the non-functional requirements. This does not imply that functional requirements are more important. Functional properties are usually the easy aspect during product derivation, and generally have to be known in order to say something meaningful about the non-functional properties.

When the engineer is unable to bind additional variation points without testing the configuration at hand, the COVAMOF Derivation Process goes to the next step, Product Realization.

9.1.3. Product Realization

In order to get a complete software product, the Product has to be realized in the product family artifacts. In COVAMOF-VS, the product is realized by pressing the Realize button in the toolbar of COVAMOF-VS. As a result, COVAMOF-VS executes the effectuation actions for each of the Variants and values that are selected for the Product entity (see also Part II).

Thereafter, Visual Studio builds the active Solution, which results in the binaries that are used together with the configuration files to test the product in the next step.

Example 11: *Let's say a product family contains the Variation Points 'VIEWPORTSIZE.HEIGHT, and 'VIEWPORTSIZE.WIDTH'. When during the Product Configuration step these variation points both have been bound to 120, the following start-up parameters will be specified in the configuration file:*

```
[Viewport]
Height=120
```

Width=120

9.1.4. Product Testing

The goal of the testing phase is to determine whether the product meets both the functional and the non-functional requirements. When, during testing, the realized product appears to satisfy all requirements, the software product can be packed and shipped to the customer. Otherwise, one or more additional iterations of a Product Configuration and Product Realization steps are required.

In any case, the values of the dynamically analyzable Dependencies that have been determined during the test are fed back into the variability model as Reference Data. In this way, the COVAMOF variability model is gradually enriched and improved to provide better estimated values during Product Configuration steps in the future.

9.2. The Benefits

COVAMOF and COVAMOF-VS were created to address the issues that are experienced by software engineers in many industrial families (see Chapter 5 and 6). In the previous section, we showed how the main entities in the COVAMOF meta-model (see also Part II) provide practicable benefits during product derivation. In this section, we summarize how the combination of COVAMOF and COVAMOF-VS addresses the variability management issues we discussed in the introduction.

1. Effectively handling complex dependencies. Instead of modeling dependencies between two variants, dependencies in COVAMOF group relations on the level of variation points. This allows specifying complex dependencies between multiple variants that would otherwise translate into a large amount of dependencies between variants. This grouping furthermore provides a more abstract view on relations between choices, thus reducing the overall complexity of the variability model. As the dependencies are first-class, COVAMOF is also able to provide a separate dependency view that shows the interaction between them.

2. Ability to use imprecise, tacit and documented knowledge. In addition to the formal specification of the variability model, COVAMOF explicitly deals with tacit, documented, and formalized knowledge through the different types of dependencies. Although tacit knowledge is not represented in a COVAMOF model (otherwise, it wouldn't be defined as tacit), COVAMOF deals with tacit knowledge by enabling references to the experts that possess this knowledge (e.g. through names, phone numbers, etc.). This may seem silly, but the importance should not be underestimated: it allows (less experienced) engineers to call-in the right assistance.

Documented and formalized knowledge are handled in several ways. The Reference Data and Associations at Dependencies enable storing useful product derivation knowledge, e.g. in terms of links to documents, graphs, and formulas. The test results for a Dependency in a particular configuration can be reused to know or estimate the value of a Dependency in a new configuration. Multiple data points can be generalized to variants with specific properties, and the results can be stored into Associations. As we explained in Part II and 9.1.2, these Associations are used during the COVAMOF Derivation Process to estimate the impact of choices on dependencies.

All these facilities allow organizations to start with a minimal amount of formalization that can pay off immediately. This model can be gradually extended when organizational maturity grows and more precise knowledge becomes available, or when more benefits are perceived for the externalization.

3. Dependency interaction. To reduce the expert involvement and number of iterations, the problem of dependency interaction is addressed by explicitly capturing Dependency Interaction entities. These entities specify a strategy that suggests to an engineer, which steps he/she should follow in trying to satisfy a particular set of interacting dependencies.

9.3. Classification of COVAMOF

Now that we have presented the modeling concepts and the derivation process of COVAMOF, we can compare our variability management framework with the techniques we presented in Chapter 6. Similar to these other techniques, we therefore explain how COVAMOF addresses the modeling and tooling aspects.

9.3.1. Modeling

The first category of our classification framework consisted of comparing how the variability is represented in the variability models. The important topics regarding this category were choices, product representation, abstraction, formal constraints, quality attributes, and support for incomplete knowledge.

- **Choices:** COVAMOF organizes the choices in a choice model. These choices are represented by Variation Point entities in the variability model.
- **Products:** The Product entities in COVAMOF model the products in a decision model.

- **Abstraction:** COVAMOF uses hierarchy as well as multiple abstraction layers for abstraction in the variability model. In addition to grouping dependencies on the level of variation points discussed in the previous section, it distinguishes between three different layers, i.e. the Feature, the Architecture and the Implementation layer. Whereas Realization Relations *within* these layers specify the hierarchical ordering of the variation point entities, Realization Relations *between* these layers specify the mapping of variability from the features, through the architecture to its implementation.
- **Formal constraints:** Similar to ConIPF and Koalish, COVAMOF use a technique-specific language, which may hamper its introduction in an organization. The constraints can contain basic logical and arithmetic operators (e.g. "or", "and", "+" and "-") on operands like variation point values and properties of selected variants.
- **Quality Attributes:** In COVAMOF, quality attributes are explicitly addressed by modeling them with the first-class dependencies. These dependencies not only discuss which choices influence the quality attributes, but also how they affect the quality attributes.
- **Support for Incomplete Knowledge and Imprecision:** In COVAMOF, the formal entities in the variability model can be extended with fuzzy rules, e.g. hints that specify directions in which dependency values will change if choices are made, or estimates based on reference data. It can furthermore contain or refer to documented knowledge (e.g. Word documents or Excel sheets on another server).

9.3.2. Tools

In the Tools category, we showed whether and how the tools of each of the techniques support the required characteristics.

- **Views:** The Visual Studio .NET tool suite of COVAMOF provides several graphical views on the variability model, i.e. two Variation Point Views, a Realization Relation View, two Dependency Views, and three Configuration Views.
- **Active Specification:** Similar to ConIPF, COVAMOF provides an editor for proactively manipulating the choices as well as constraints in the product family artifacts. COVAMOF is supported by a proprietary Visual Studio Add-in that is designed for drawing the choices and relations in a graphical view of the variability model. This tool prevents inconsistency, for example, by preventing the specification of constraints between choices that do not exist, and preventing the deletion of choices when there are still constraints referring to them.
- **Configuration Guidance:** Similar to ConIPF, procedural knowledge is used in the configuration tool of COVAMOF. The tool describes strategies to realize

certain functional and non-functional properties of the system, based on the choices in the variability model. It furthermore dynamically advices a configuration strategy based on statistics, such as the number of choices associated to particular constraints.

- **Inference:** COVAMOF provides, in addition to consistency checking, an inference engine that automatically makes decisions based on the rules that are specified in the realization relation and the dependencies, and the partial configuration.
- **Effectuation:** The Variation Point entities in COVAMOF variability models specify the actions that have to be performed based on how they are configured for a product. COVAMOF-VS can automatically execute these actions from the configuration interface. These actions encompass the specification of pre-processor settings and language constructs for C, C++ and C# projects, the generation of configuration files and other file-based operations. Like Pure::Variants and VSL, companies can also easily extend COVAMOF-VS with additional types of effectuation actions.

As we showed in this classification, COVAMOF supports all characteristics of the modeling and the tools aspects. To compare COVAMOF with the other five variability modeling techniques, we extend Table 13 with COVAMOF in the table below (Table 14, page 158).

9.4. Conclusion

In this chapter, we discussed the COVAMOF Product Derivation Process. We described this process using the technical realization of COVAMOF in the form of the COVAMOF-VS tool-suite. We have also shown how the different elements of COVAMOF address the key variability management issues during product derivation and compared COVAMOF with the techniques described in Chapter 6. In the following chapter, we present the empirical results from an experiment with the industrial application of COVAMOF.

Table 14. The classification of COVAMOF and the five variability modeling techniques we presented in Chapter 6.

	Approaches / Characteristics	VSL	ConIPF	CBFM	Koalish	Pure::Variants	COVAMOF
Model	Choices	Choice model (Variabilities and Variation Points)	Multiplicity in structure (Feature and Artifact tree)	Multiplicity in structure (Feature tree)	Multiplicity in structure (Koala components)	Multiplicity in structure (Feature and Family model)	Choice model (Variation Points)
	Products	Decision Model	Decision Model	Decision Model	Stand-alone entity	Decision Model	Decision Model
	Abstraction	Multiple Layers	Hierarchy and Multiple Layers	Hierarchy	Hierarchy	Hierarchy and Multiple Layers	Hierarchy and Multiple Layers
	Formal Constraints	Include/exclude (technique specific)	Algebraic expressions (technique specific)	Algebraic expressions (XPath)	Algebraic expressions (technique specific)	Algebraic expressions (Prolog)	Algebraic expressions (technique specific)
	Quality Attributes	Not modeled	Mentioned (Context entities, restrictions)	Mentioned (Feature entities)	Not modeled	Not modeled	Modeled (Dependency entities)
	Support for Incompleteness and Imprecision	Requires precise specifications	Aimed at completeness, requires precise specifications	Requires precise specifications	Requires precise specifications	Aimed at completeness, requires precise specifications	Supports both
Tooling	Multiple Views	Not supported by tooling	2 (modeling & configuration)	2 (modeling & configuration)	1 (configuration)	4 (modeling & configuration)	8 (modeling and configuration)
	Active Specification	Not supported by tooling	Proactive	Reactive specification of constraints	None	Proactive is possible, no consistency checking	Proactive
	Configuration Guidance	Not supported by tooling	Static and dynamic	None	None	None	Static and dynamic
	Inference Engine	Not supported by tooling	Consistency, decision making, prevention	Consistency	Consistency, decision making, prevention	Consistency, decision making	Consistency, decision making
	Effectuation	Operations on XML documents and Jscript sources.	None provided.	None provided.	Generation of C applications.	File-based operations with special support for C and C++.	File-based operations with special support for C, C++, and C#

Chapter 10 Validation of COVAMOF during Product Derivation

COVAMOF is a variability management framework for product families that was developed to reduce the number of iterations required during product derivation and to reduce the dependency on experts. In this chapter, we present the results of an experiment with COVAMOF in industry. The results show that with COVAMOF, engineers that are not involved in the product family were now capable of deriving the products in 100% of the cases, compared to 29% of the cases without COVAMOF. For experts, the use of COVAMOF reduced the number of iterations by 42%, and the total derivation time by 38%.

Based on	Section numbers
M. Sinnema, S. Deelstra, Industrial Validation of COVAMOF, Elsevier Journal of Systems and Software, Vol 81/4, pp. 584-600, 2007.	All sections in this chapter

10.1. Introduction

Variability, or the ability to be subject to change, customization, configuration or extension for use in a specific context, is introduced in the reusable artifacts (such as an architecture or components) of a software product family (Bosch, 2000; Clements and Northrop, 2001; v.d. Linden, 2002; Weiss and Lai, 1999) to be able to derive different products. In Chapter 3, we identified that organizations employ a generic process for deriving these products. In this process, organizations develop a first configuration of their product during the initial phase, which is subsequently modified in the iteration phase until the product meets the requirements (see Figure 47 for a high level view on this process).

In a case study at Thales and Bosch (see Chapter 4), we identified the problems that organizations face during this process of product derivation. Two of the main problems are (1) that product derivation is too dependent on experts, and (2) that too many iterations are required before a product is finished. These iterations have a great impact on cost and time-to-market.

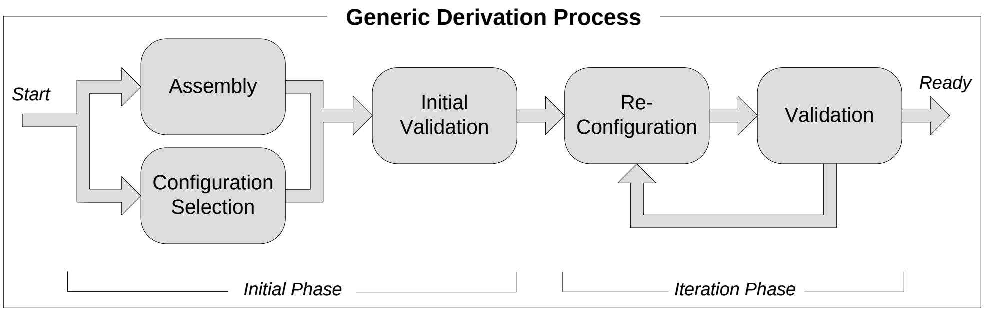


Figure 47. Product derivation in software product families. A typical product derivation process starts with an initial phase, where a first configuration is constructed using assembly or configuration selection. The process subsequently goes through a number of iterations before the product is finished.

The problems during product derivation are mainly caused by complexity and implicit properties, which makes it hard to manage the variability in the product family artifacts (see Chapter 5). An approach to tackle these issues is variability modeling. Variability modeling aims at externalizing (Nonaka and Takeuchi, 1995) the required variability information, in such a way that engineers can be supported by tooling.

A fundamental problem in this context, however, is that it is not always possible or worthwhile to externalize all variability information to a fully formalized variability model. First of all, externalization is expensive and time consuming, as it requires investment in an infrastructure and maintenance, and intervenes with everyday business. Second, externalization suffers from the law of diminishing returns (Spillman and Lang, 1924), which suggests that at some point, formalization is unprofitable with respect to the costs and risks of leaving information tacit and documented. Third, the information that is available is often incomplete or imprecise. Think for example of quality attributes, where experts sometimes only know which choices affect them, but only in terms of estimates based on average behavior, or in terms of directions in which a value will change when a choice is reconsidered.

In Chapter 6, we discussed why other variability modeling techniques, such as Asikainen et al. (2004), Becker (2003), and Czarnecki et al. (2005), do not address these issues sufficiently. When we started the development of our variability management framework COVAMOF in 2002, we therefore explicitly designed it to deal with both precise formalized knowledge and incomplete and imprecise documented knowledge. To validate our approach, we performed several experiments in industrial organizations.

Experiments in the context of the ConIPF project (Configuration of Industrial Product Families, Hotz et al., 2006) already showed the benefits of partially formalizing variability information. In these experiments, for example, Thales Naval Netherlands validated a proprietary method for modeling variability. Their method used a subset of the COVAMOF formalisms, i.e. it only focused on specifying valid ranges, valid combinations of variation points, and requires/exclude relations. Despite the fact that this is a smaller scope than our COVAMOF method, the experiments performed during the ConIPF project (Hotz et al., 2006) already show a large improvement over not using variability modeling at all.

As the contents of the experiment reports are confidential and the experiment did not address the entire scope of COVAMOF, we started an additional series of experiments to test how well COVAMOF performs. These experiments were conducted at Dacolian B.V. (Dacolian B.V. website), worlds leading supplier of OEM software modules for intelligent traffic systems that utilize automatic license plate recognition (ALPR). The experiments thus involve an industrial product family at an organization that was not part of the ConIPF project.

This chapter presents the results of one of these experiments. We will show that, for experts, the use of COVAMOF reduced the number of iterations required to derive products by 42%, and the total product derivation time by 38%. In addition, we will show that while nonexperts were only able to derive products in 29% of the cases without COVAMOF, they succeeded in 100% of the cases with COVAMOF.

To present the results of the experiment, we have structured this chapter as follows. In the next section, we discuss the hypotheses that were tested during our experiment. In Section 10.3, we discuss the context of the experiment, i.e. the product family, and the products. In Section 10.4, we describe how we conducted the experiment, and in Section 10.5, we describe the results. We evaluate the results in Section 10.6, present experiences of the participants in Section 0, and conclude this chapter in Section 10.8.

10.2. Hypotheses

COVAMOF was primarily developed and evaluated based on the Bosch and Thales case, within the settings of the research project Configuration of Industrial Product Families (ConIPF). To improve the validation of COVAMOF, we wanted another industrial test case, where different results might be achieved, and new (unforeseen) issues could arise.

As we indicated in the previous section, the experiment we describe in this chapter does not focus on validating COVAMOF as a whole, but we explicitly chose an incremental approach instead. Rather than validating whether setting up, using, and maintaining the COVAMOF approach in a product family reduces total cost, or increases profit (either through improved quality, time-to-market or effort saved), we decided to first experiment on two facets, i.e. (1) the effort required during product derivation, and (2) expert dependency. Below, we formulate the two hypotheses that are connected to these facets.

Hypothesis 1. The first hypothesis involves product derivation effort. Our hypothesis is that, in terms of being able to derive a product that adheres to the product requirements, experts should not perform worse when using COVAMOF. In other words, if experts were able to derive a product without COVAMOF, they will also be able to derive them with COVAMOF. In addition, on average, the effort required to derive these products will be less when using COVAMOF, than without the use of a variability management technique (i.e. their everyday practice).

Hypothesis 2. The second hypothesis involves the dependency on experts during the product derivation process. Our hypothesis is that with a COVAMOF model of a product family in place, the product derivation process will be less dependent on experts. More precisely, with such a model in place, engineers that are not involved in the development or derivation of the product family will also be able to derive products. Similar to Hypothesis 1, they will, on average, require less effort in deriving those products when they use COVAMOF.

Note that, for Hypothesis 1, we do not test whether COVAMOF helps in situations where none of the experts would be able to derive a product that adheres to the requirements. Furthermore, benefits such as the possibility to incrementally build up the COVAMOF model, the reduction of complexity, or the benefits it brings to evolving the product family are also not (directly) measured through this experiment.

Although our experiment does not deal with determining the strength of the creation process of the variability model, product derivation with COVAMOF is still dependant on this model. We therefore make the following assumption.

Assumption. It is possible to create a COVAMOF model that is useful during product derivation.

In Section 10.4, we discuss the relation between the set-up of the experiment and the validation of the hypotheses and assumptions.

10.3. Industrial Context

The goal of the experiment was to validate whether COVAMOF improves the product derivation process in practice. To ensure that this experiment would be representative, we required its context to be industrially realistic. Similar to the previous experiments (Hotz et al., 2006), we therefore again chose to perform this validation with engineers from an industrial organization. Furthermore, the product family had to be highly complex in terms of size, target domain and technology. This is why we selected Dacolian and its Intrada product family we introduced in Part II.

Dacolian faces similar problems as the ones we identified in Chapter 5. The purpose of the experiment was therefore twofold. On the one hand, we wanted to validate the method we developed over the past few years. On the other hand, Dacolian wanted to determine whether COVAMOF was an appropriate method that they could, primarily, use to solve their problem of dependency on experts.

Due to the size of the Intrada product family, an experiment on the entire family would be too expensive, particularly given the fact that the purpose of the experiment was to determine whether COVAMOF is effective. Naturally, the experiment had to be industrially realistic in order to be convincing. As Dacolian maintains the individual modules of their entire product family separately, each individual module of the Intrada product family can be viewed as a product family. In this view, those configured modules are used as products that are used to develop the Intrada systems. One of those product families is the FindComponents (FC) module, which contains all variability realization techniques that can be found in the rest of the system. Together with Dacolian, we determined that this product family was particularly suitable for an industrially realistic experiment.

The FindComponents (FC) products perform the most complex and important step in all Intrada systems, as it is responsible for determining whether the input image contains a license plate, as well as the initial separation between the background and foreground pixels in the input images, e.g. separating characters from the background. Important quality attributes that are associated to this product are, for example, the average processing time required to process images, the memory usage, or the percentage of images that are incorrectly rejected.

The input images and importance of quality attributes differ significantly for different customers. The type of input images (e.g. low noise, high contrast, whether a large part of the environment is visible, etc.) has a large impact on how a configuration scores on the quality attributes. In most cases, the same configuration scores far better or worse on different types of input images, and may be acceptable to one customer, but unacceptable to the other.

The validation steps of an FC configuration break down into testing its performance on a representative set of input images. These tests require about 40 min on one machine.

10.4. Experiment Description

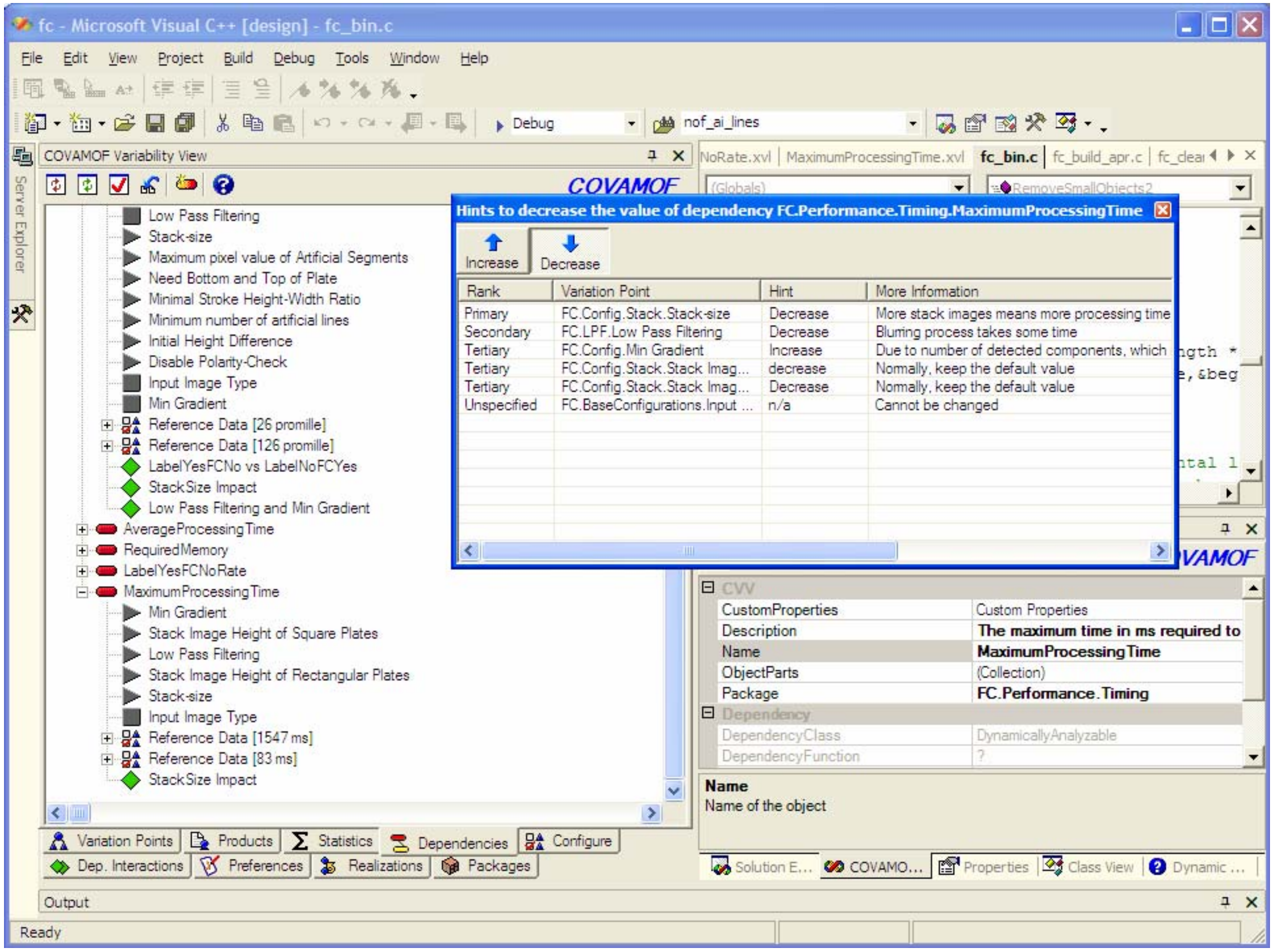
Now that we have presented the COVAMOF variability management framework, our hypotheses, and the context of the experiment, it is time to get to the core of this chapter. In this section, we describe the set-up of our experiment.

10.4.1. COVAMOF Model of the FindComponents Product Family

As we described in Section 10.2, the hypotheses were tested in the context of an existing COVAMOF model. To create this model, we used the externalization process that we will describe in Chapter 12. In short, this means we sat down with the *Component Developer*, did code inspections, studied documentation, and tested several FC configurations, to identify the variation points and dependencies. The variation points and dependencies were subsequently inserted into the FC artifacts, and several of the existing FC configurations were used to test the model for errors. We also used additional interviews to create hints for dealing with dependency interactions. Figure 48 shows a screenshot of this model in the COVAMOF-VS tool-suite, and in Figure 49 we present a small example from the COVAMOF model of the FindComponents product family.

10.4.2. Requirements-Sets

To be able to judge the difference between product derivation with and without COVAMOF, we needed more than one Requirements-Set to experiment on. A Requirements-Set contains the functional and non-functional requirements for one product. Obviously, a participant would be able to reuse the knowledge gained if he or she would be required to experiment with and without COVAMOF on the same Requirements-Set. With the *Component Developer*, we therefore selected three Requirements-Sets for products in the different product domains we described in the Dacolian B.V. case study in Chapter 4, i.e. Tolling, LawEnforcement and Parking. The Requirement-Sets were selected from the large number of Requirements-Sets of products that have been derived in the past. An example of a non-functional requirement in a Requirement-Set was the restriction on the “StackSize” as we showed in Figure 49. These Requirements-Sets were different from the requirements of the configurations that were used to test the COVAMOF model.



Rank	Variation Point	Hint	More Information
Primary	FC.Config.Stack.Stack-size	Decrease	More stack images means more processing time
Secondary	FC.LPF.Low Pass Filtering	Decrease	Blurring process takes some time
Tertiary	FC.Config.Min Gradient	Increase	Due to number of detected components, which
Tertiary	FC.Config.Stack.Stack Imag...	decrease	Normally, keep the default value
Tertiary	FC.Config.Stack.Stack Imag...	Decrease	Normally, keep the default value
Unspecified	FC.BaseConfigurations.Input ...	n/a	Cannot be changed

CVV	
CustomProperties	Custom Properties
Description	The maximum time in ms required to
Name	MaximumProcessingTime
ObjectParts	(Collection)
Package	FC.Performance.Timing
Dependency	
DependencyClass	DynamicallyAnalyzable
DependencyFunction	?

Name
Name of the object

Figure 48. COVAMOF-VS screenshot. This screenshot shows an excerpt of the FindComponents variability model in the COVAMOF-VS Add-In for Microsoft Visual Studio.

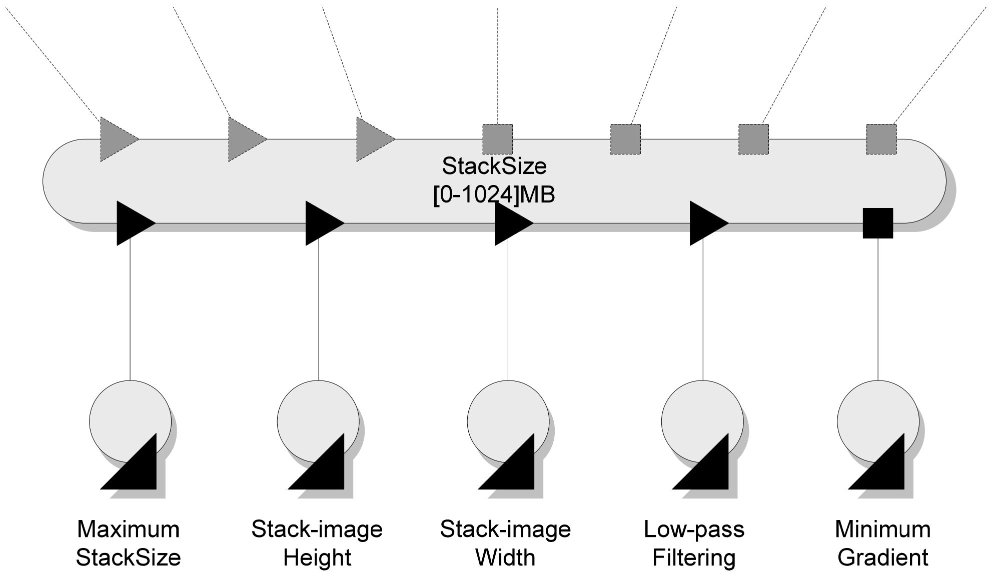


Figure 49. Example from the COVAMOF Model of the FindComponents product family. The value of the Dependency "StackSize" depends, amongst others, on the selected Maximum StackSize, Stack-image Height, Stack-image Width, the number of Low-pass Filtering steps and the selected Minimum Gradient. The first three variation points involve the storage of temporary images and the other two variation points involve aspects of the image processing steps. Whereas the direction of the impact of the first four variation points is known in advance (Directional Association), the direction of the impact of the Minimum Gradient variation point is not known (Abstract Association). The only information available on this last variation point is that its selected value does have impact on the Required Memory.

Due to the completely different physical properties of the systems in these domains, each Requirements-Set requires significantly different configurations. In addition, the knowledge required for, and gained from deriving the product in one domain is not useful for deriving products in the other domains. With knowledge gained we mean: insights that might be acquired, for example, from the effects of changes to variation points on quality attribute values, during several iterations. The insights gained during the iterations for one Requirements-Set would therefore not be applicable for the other Requirements-Sets. In other words, the numbers of iterations that are measured are not influenced by the fact whether a product for another Requirements-Set has been derived before.

The hundreds of products that have been derived over the past few years furthermore show that the products that have been derived by the same engineer typically require the same amount of iterations. This implies that, despite the fact that the three Requirements-Sets are significantly different, the results of experiment on different Requirements-Sets can be compared to each other. The

Component Developer required three iterations to derive each product for the Requirements-Sets we selected.

10.4.3. Participants

Besides ourselves, three types of participants were involved in the experiment:

1. A person that was responsible for developing and maintaining the FC product family assisted in setting up the experiment. We refer to this person as a *Component Developer*.

2. To test Hypothesis 1, i.e. whether COVAMOF would speed up product derivation for experts, we invited four software engineers from Dacolian. These engineers had not been involved in deriving the original products for the Requirements-Sets from the Tolling, LawEnforcement and Parking domain. We refer to these participants as *Experts*. The level of experience of each engineer differed, however. While two engineers are considered domain experts at Dacolian, the two other engineers would typically require a lot of support from a *Component Developer* during product derivation. Note that during the experiment, none of the participants was allowed to consult the *Component Developer*.

3. To test Hypothesis 2, we invited five experienced software engineers that did not work at Dacolian, that had the same level of education and experience, and that had no prior knowledge of the FC product family. We refer to these participants as *NonExperts*.

Except for the *Component Developer*, and one *NonExpert*, all participants were available for two full person days. COVAMOF was a new technology for all participants.

Before conducting the experiment, we therefore presented the ideas behind the COVAMOF variability modeling framework. In addition, each participant received a tutorial for the COVAMOF-VS tool-suite.

10.4.4. Mapping Participants to Requirement-Sets and Product Derivation with and without COVAMOF

For the experiment, we had two hypotheses, and one assumption that needed to be validated. To validate these hypotheses and assumptions, we mapped the participants to the Requirement-Sets and product derivation as follows:

Experts

The *Experts* were required to derive products for the Requirements-Sets; once or twice without COVAMOF, and once or twice with COVAMOF. The situations without and with COVAMOF were distributed over the Requirements-Sets. This distribution is described in Table 15.

For product derivations without COVAMOF, the *Experts* were allowed to study the source code. The order of the experiment was first without COVAMOF, and then with COVAMOF. Note that, since the knowledge gained during the iterations without COVAMOF is not useful due to the significant difference in problem domain, this order does not influence the number of iterations when performing the experiment with COVAMOF.

NonExperts

The set-up of the experiment for *NonExperts* was equal to the *Experts*. In addition to the set-up described for the experts, however, we instructed one *NonExpert* to perform all product derivations with COVAMOF (Participant 5). The distribution for the *NonExperts* is also described in Table 15.

Table 15. This table lists, for each participant (numbered 1–9 in the second column of each row), in which sequence (1, 2 and 3) products from the three Requirements-Sets were derived. The COVAMOF column indicates whether the participant used COVAMOF for the associated Requirement-Set. An "×" in the table means that the participant did not have time to perform an experiment with the associated Requirement-Set.

		Requirements Set 1		**Requirements Set 2**		**Requirements Set 3**	
		Sequence	**COVAMOF**	**Sequence**	**COVAMOF**	**Sequence**	**COVAMOF**
Experts	1	3	Yes	1	No	2	No
	2	1	No	3	Yes	2	No
	3	1	No	2	Yes	3	Yes
	4	3	Yes	1	No	2	Yes
NonExperts	5	2	Yes	3	Yes	1	Yes
	6	1	No	2	No	3	Yes
	7	3	Yes	1	No	2	Yes
	8	1	No	×	×	2	No
	9	2	No	3	Yes	1	No

With this mapping, we consider the hypothesis and assumptions validated under the following conditions:

Validation for Assumption: If the experiment proves both hypotheses, viz. if the experiment proves that *Experts* and *NonExperts* are able to derive products faster, it also proves it is possible create a COVAMOF model that is useful during product derivation.

Validation for Hypothesis 1: The validation of Hypothesis 1 is straight-forward. The experiment results of the *Experts* should show that with COVAMOF, they will succeed in deriving products, and, on average, require less iterations for product derivation.

Validation for Hypothesis 2: The validation of Hypothesis 2 is also straight-forward. The experiment results of the *NonExperts* should show that with COVAMOF, they will succeed in deriving products, and, on average, require less iterations for product derivation.

The measurements that are used to validate these hypotheses are discussed in Section 10.4.6.

10.4.5. Derivation during the experiments

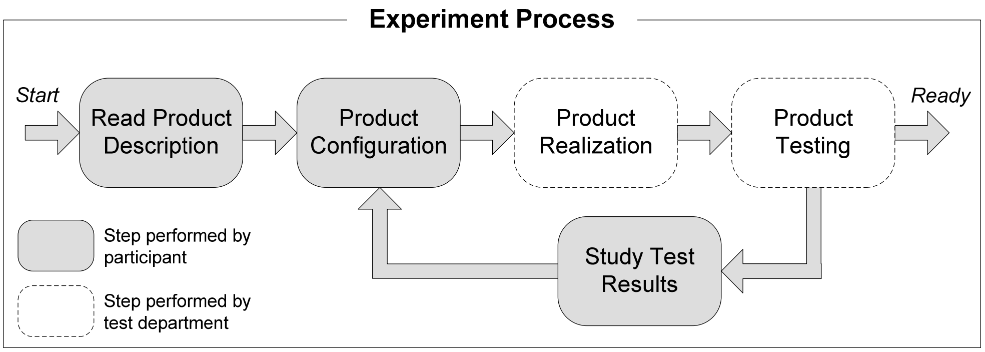


Figure 50. Product derivation during the experiments. This figure depicts the steps the participant took during the experiment. The product configuration step either consisted of configuration with or without COVAMOF. Compilation and testing was performed by the test department.

During the experiment, the participants followed the generic product derivation process as described in Section 10.1, either by following the normal procedure at Dacolian, or by following the COVAMOF Derivation Process. In these derivation processes, engineers first go through the initial phase, and then require zero or more cycles through the iteration phase. The initial phase and iteration cycles are each concluded with a validation step. In the process picture for the experiment, the validation step is split in two steps: product testing, and study test results (see Figure 50). For the experiment, the product realization and testing steps were performed by the test department. The product realization and product testing steps thus defines the end of an iteration cycle for a participant.

10.4.6. Measurements

We were interested in determining the benefit that COVAMOF brings to the derivation process. As COVAMOF focuses on assisting the engineer in things such as selecting the right components and setting the right parameters, we were primarily interested in the difference between effort spent in steps that deal with these choices, and whether *NonExperts* were able to derive products when using COVAMOF.

For the experiment, the product testing step determined whether the participant succeeded in deriving the product that adheres to the given Requirements-Set (see Figure 50). To this purpose, each time the participant wanted to test whether the product was finished he had to hand the configuration to the test department.

To measure the difference in effort, we focused on two measurements, i.e. the number of hours, and number iterations that were required by the participants. Each time a configuration was handed to the test department, we counted one iteration cycle. We used a stopwatch to measure the time. The participants were furthermore monitored while deriving a product. This allowed us to separate the hours spent on tool-usage issues and actual derivation effort, and to collect unexpected, but interesting observations.

As we expected that the *NonExperts* would not always be able to derive an acceptable FC module, we set a maximum number of hours a participant could spend on derivation. This maximum was set to one iteration more than two times the number of iterations that originally had been required for the Requirements-Sets. This maximum of seven iterations was chosen, as this was the maximum number of iterations that Dacolian was willing to accept to let *NonExperts* derive their products. When participants required more time, the result was noted as ‘Was unable to derive an acceptable product’.

In theory, the participants might have been able to derive the products when they would have been allowed to continue. When a participant is unable to derive a product within seven iterations, we therefore use eight iterations in our calculations. This is the theoretical minimum number of iterations they would require in case we would have allowed them to continue.

This experiment design affects our conclusions in two ways. First, where we calculate the percentage of participants that were not able to derive a product, the actual numbers could be higher; theoretically they could have been able, but needed more iterations. Like we discussed above, however, in order for COVAMOF to be useful to Dacolian, the maximum number of acceptable iterations was seven, which is why calculating the percentage in our way was still useful.

The second way in which this experiment design affects the conclusions is when we calculate the improvement in terms of number of iterations. The conclusions have to take into account that if eight iterations were measured, the actual number could be higher.

10.5. Experiment Results

In a period of four weeks we successfully conducted the experiment described above. Throughout this section, we present the results of this experiment. We start with presenting the raw quantitative data in terms of required iterations and hours. We furthermore present our observations and remarks from the participants. Finally, we explain how these results support the hypotheses and assumptions we presented in previous sections.

10.5.1. Raw Data

Like we described in Section 10.4.6, we logged for each participant, the number of iterations, and number of hours they required to derive a product that fulfilled the requirements. Figure 51 to Figure 54 represent the quantitative results from each individual experiment. Figure 51 and Figure 52 show the number of iterations required by the participants, while Figure 53 and Figure 54 show the amount of hours required during these iterations. In both figures, we numbered the participants from 1 to 9, where the first four participants are the *Experts* and the other five participants are the *NonExperts*. Note that we counted the situations in which participants failed to derive a product within seven iterations as requiring eight iterations (see also Section 10.4.6).

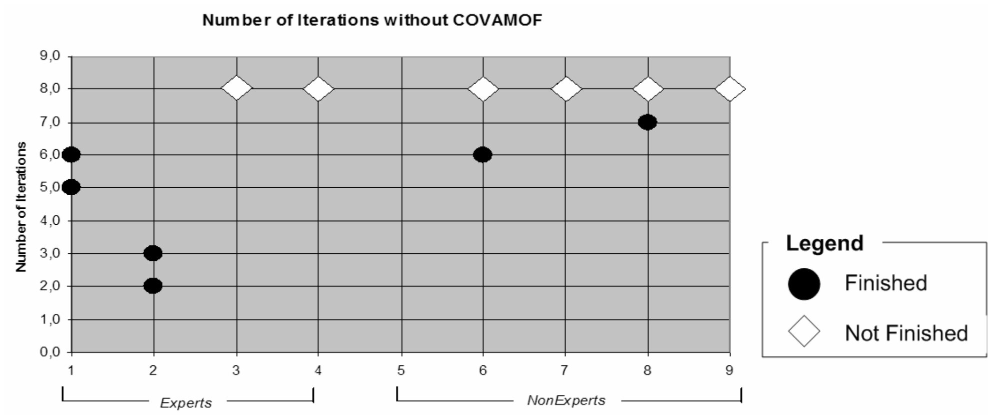


Figure 51. The number of iterations measured during the experiments without COVAMOF. Each vertical line corresponds to a participant and each marker indicates the number of iterations required in one experiment. Each experiment that did not result in a successful configuration after seven iterations is counted as requiring eight required iterations.

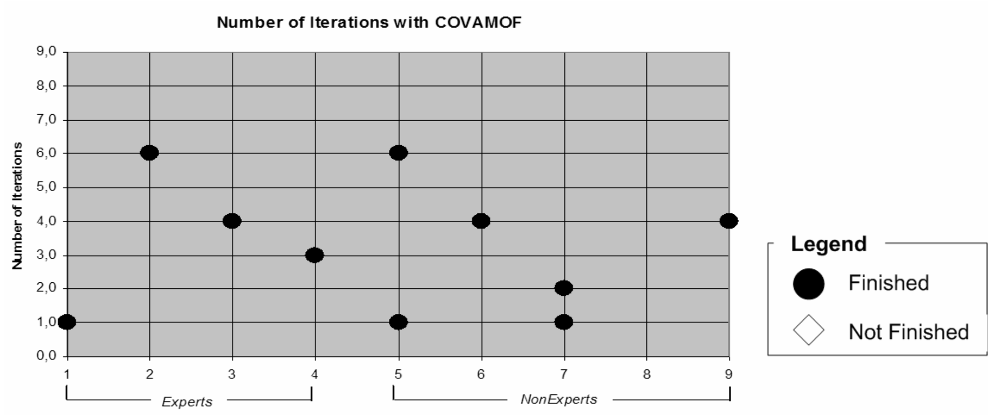


Figure 52. The number of iterations measured during the experiments with COVAMOF. Each vertical line corresponds to a participant and each marker indicates the number of iterations required in one experiment. Each experiment that did not result in a successful configuration after seven iterations is counted as requiring eight required iterations.

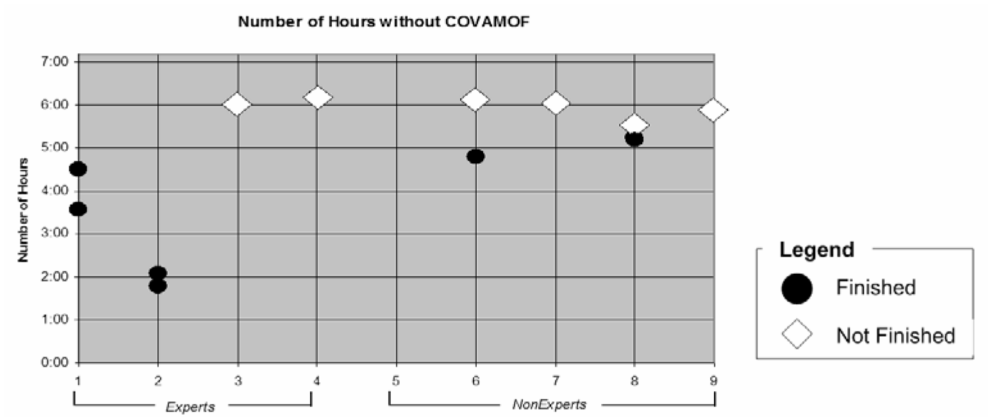


Figure 53. The number of hours measured during the experiments without COVAMOF. Each vertical line corresponds to a participant and each marker indicates the number of hours required for deriving one product.

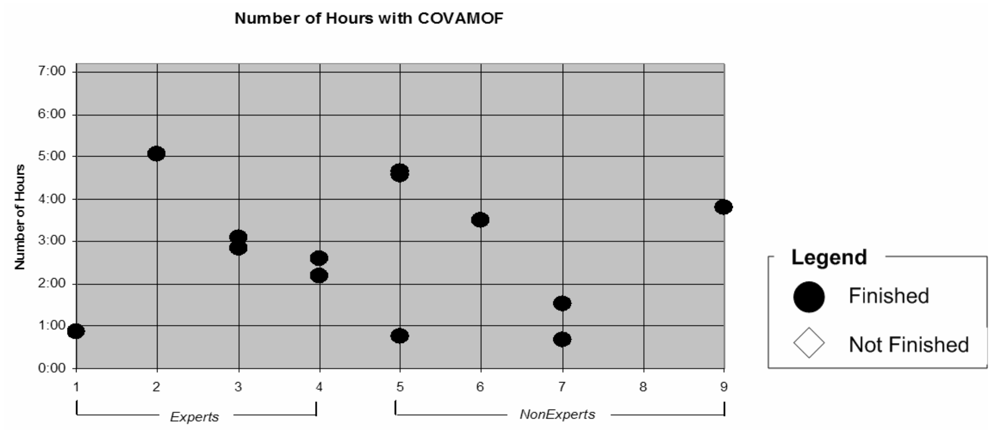


Figure 54. The number of hours measured during the experiments with COVAMOF. Each vertical line corresponds to a participant and each marker indicates the number of hours required for deriving one product.

10.5.2. Additional Observations

In addition to the hours and iterations, we also logged our observations. The observations focused on the way participants came to a new configuration, in particular how this changed after the introduction of COVAMOF.

- Figure 51 shows that two of the experts (Participants 3 and 4) performed worse than the other two experts when deriving products without COVAMOF. This reflects the difference in level of expertise of these experts, and the fact that these two experts were the ones that would normally require assistance from a *Component Developer* (see also Section 10.4.3).
- With COVAMOF, the experts paid less attention to the source code of FC. Instead, they were more relying on the hints they received from COVAMOF. This emphasizes their confidence in the fact that the COVAMOF model would bring them more benefits than the source code.
- A third observation is that being an expert can be a drawback. From practice, we knew that several experts can have conflicting insights; where one would think the effect of a change would be one way, another expert can think the opposite. While we take these effects into account when we create COVAMOF variability models, we now also experienced them first hand during the product derivation process. Some experts would first deviate from the hints provided by COVAMOF, as they believed from their own frame of reference that they should do otherwise. When studying the test results, they were surprised to see that the original hints were in fact correct. The effect of this insight is twofold. Obviously, if they would have been aware of the fact that their insights might be wrong, some of the experts could have scored better with COVAMOF. A second important effect of the insight by the participants is that performing product derivation with this model improves knowledge sharing, as differences between insights of experts now become visible.
- A fourth observation was the different ways in which participants tackle a configuration problem, which is one of the reasons for the differences between results measured for different participants. This was clearly visible both with, and without COVAMOF. On the one hand, for example, COVAMOF-VS calculates a ranking of VariationPoints that based on the Impact property of Associations (see Part II). This is the most likely best order in which the selection at VariationPoints should be changed. One participant would instead first exclude a few of the other possibilities prior to following the order suggested by COVAMOF. On the other hand, without COVAMOF, some participants would rigorously change choices to determine the effect of changes, while others would only marginally change them.
- A final interesting observation is that in some cases the participants that used COVAMOF created configurations that scored much better on the quality attributes then the configurations that originally had been constructed for the Requirements-Sets.

10.6. Evaluation

We also derived additional views based on the raw data. Each distinguishes between participants from the Experts and from the NonExperts group. We use these views to evaluate our hypotheses and assumptions.

10.6.1. Percentage of participants that finish successfully

The first interesting view is the percentage of participants that finished successfully, and how this percentage changed after the introduction of COVAMOF. In Figure 55, these percentages are visualized. This figure shows that without COVAMOF 33% of the *Experts* and 71% of the *NonExperts* did not finish up with an acceptable configuration, whereas with the support of COVAMOF all Experts as well as all *NonExperts* do succeed in this task. These results already show the major benefit of COVAMOF for *Experts* as well as *NonExperts*, as well as the validation for the Assumption.

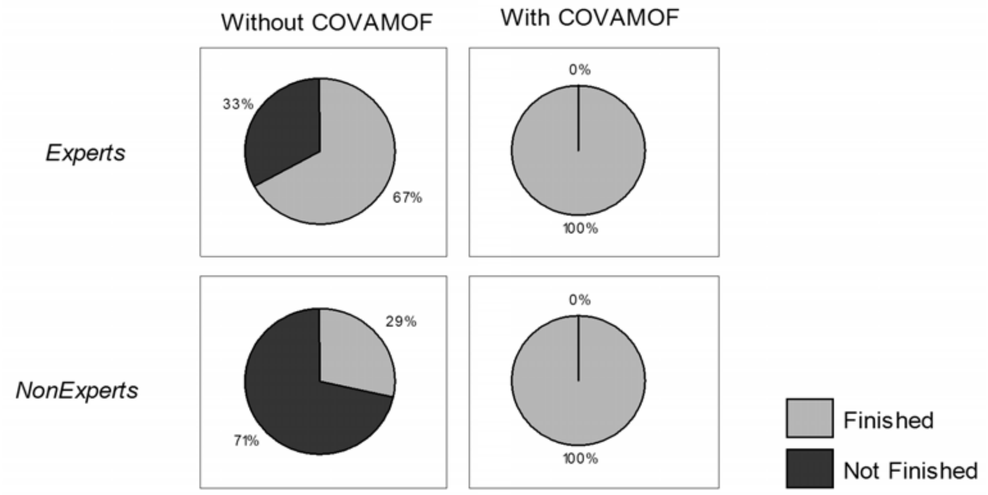


Figure 55. Percentage of participants that finished. This figure shows the effect of COVAMOF on the percentage of participants that are able to finish successfully within seven iterations.

10.6.2. Number of Iterations

The second interesting view is the reduction of the average number of iterations required by the participants, as well as the maximum and minimum number of iterations. These numbers are visualized in Figure 56. For the graphs in that figure, we calculated the average number of iterations and hours of each individual participant. Based on these averages, we calculated the average minimum and maximum numbers for all of the participants (grouped as *Experts* and *NonExperts*).

Where for the *Experts*, this graph already shows an average reduction of 2.5 iterations, it shows for *NonExperts* that COVAMOF dramatically reduces the average number of iterations by more than 4 (from 7.6 to 3.5 iterations). Together with the information from Figure 55, we can conclude that these experiment results thus firmly support Hypothesis 1 and 2.

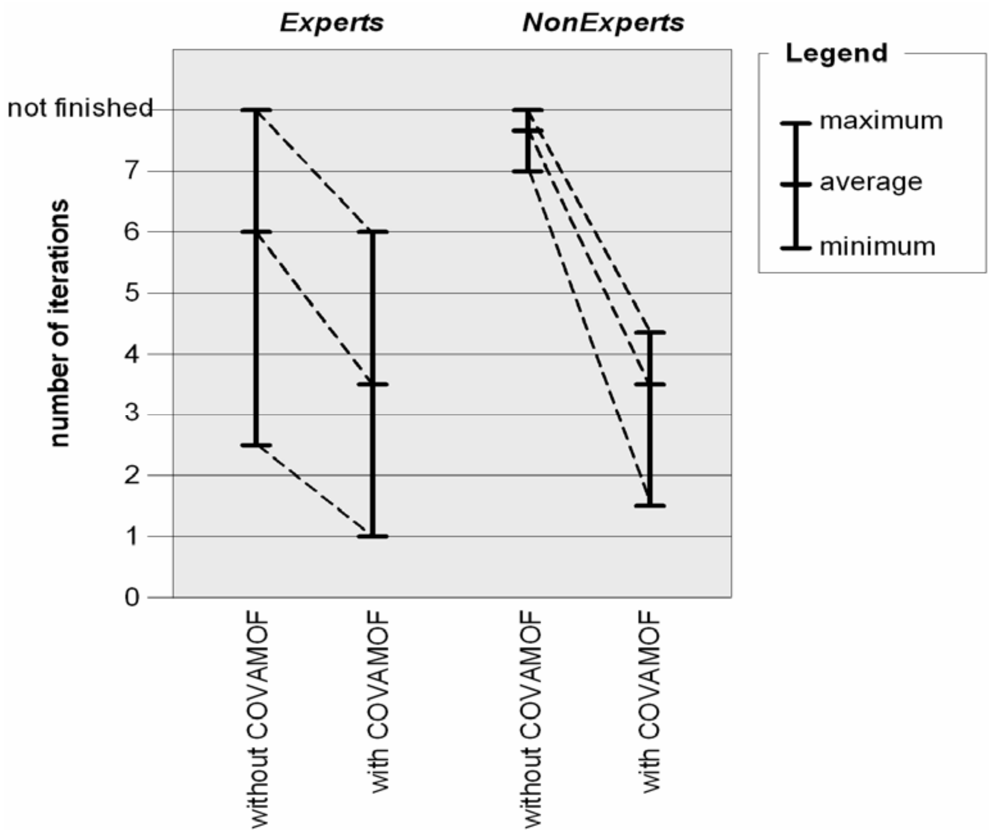


Figure 56. Number of iterations that were required to derive the products. This figure shows the effect of COVAMOF on the number of iterations. Note that we counted each participant who did not finish after seven iterations as finished after eight iterations.

Please note that only without the use of COVAMOF, situations occurred in which the participants did not finish within seven iterations. The improvements of COVAMOF we present in this chapter thus represent a theoretically minimum improvement; we use eight iterations in our calculations for those situations, while the actual number of iterations could be higher, and thus also the improvement (see also Section 10.4.4).

The effects of a reduction in number of iterations are significant. A smaller number of iterations, for example, leads to a faster time-to-market, saves production costs when less prototypes need to be manufactured, and requires less person-hours for testing. In the following sections, we focus on the benefits in terms of time. We divide this discussion in the total time required for derivation, and the total time spent in the configuration step.

10.6.3. Total Number of Hours

The third interesting view is the average total time required by the participants. These numbers are visualized in Figure 57, together with the maximum and minimum of these numbers. For these graphs, we calculated the average number of hours of each individual participant with and without COVAMOF. Based on these averages, we calculated the minimum, average and maximum numbers for all of the participants (grouped as *Experts* and *NonExperts*).

The improvement in number of hours required by the *Experts* is about one and a half hours, from an average of 4:32h to 2:49h. For the *NonExperts*, this reduction is even higher. The total number of hours for this group is reduced by more than two and a half hours from an average of 5:42h to 2:56h.

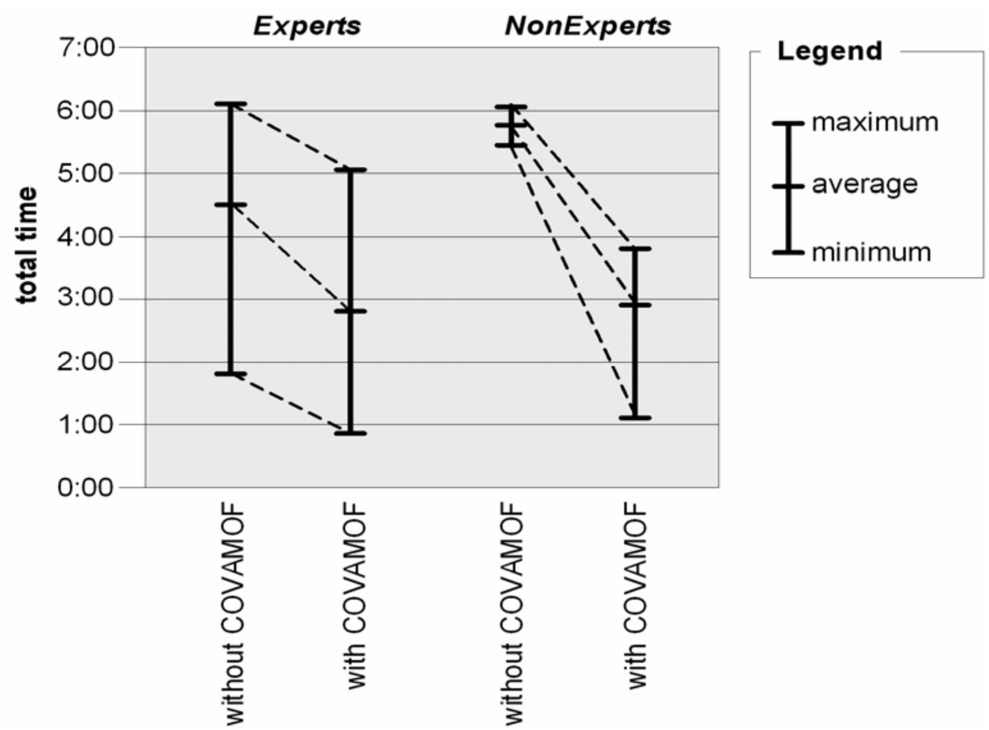


Figure 57. Total time required to derive the products. This figure shows the effect of COVAMOF on the total number of hours spent.

The total time required to derive a product is in fact a function of the configuration time, and the testing time:

$$total\_time = \#iterations \times (config\_time + testing\_time) \qquad (1)$$

Due to the large difference between the configuration time and testing time, the effects of a decrease in the number of iterations are much larger than a decrease in configuration time (see also Section 1.1.1). To illustrate the impact of testing time, Figure 59 visualizes how the benefit of COVAMOF changes with the testing time (see also Equation 1). The dashed lines in the figure refer to the total time required by *Experts* and *NonExperts* configuring in the original setting, while the solid lines refer to the total time required by *Experts* and *NonExperts* configuring with COVAMOF. The vertical line indicates the testing time in our experiment, i.e. 40 min.

Although the improvement *factor* between the total time with and without COVAMOF hardly changes, the absolute time, increases dramatically. For example, with a testing time of three hours, a *NonExpert* would require almost 24h without COVAMOF, while he or she would already finish within 12h with COVAMOF. In other words, the benefit of COVAMOF becomes even more apparent with larger testing times.

10.6.4. Time Required for Configuration

To dig deeper into the relationship between configuration time, testing time, and number of iterations, we can also use our measurements to see how the configuration time changes after the introduction of COVAMOF (see Figure 58). We call the change in configuration time from the situation without COVAMOF to the situation with COVAMOF, the ConfigTimeFactor (see also Equation 2). Thus, when COVAMOF would have no effect on the configuration time, the ConfigTimeFactor is exactly 1:

$$ConfigTimeFactor = \frac{config\_time_{COVAMOF}}{config\_time_{original}} \qquad (2)$$

The first observation that follows from Figure 58 is the difference between *Experts* and *NonExperts*. Where for *Experts*, we measured a reduction of the configuration time, for *NonExperts* this time increases significantly. The cause of this difference most likely lies in the amount of existing background knowledge of the participants. As the *NonExperts* lacked relevant background knowledge, without COVAMOF, they oftentimes resorted to guessing during the configuration step. With COVAMOF, however, they required time to study the model.

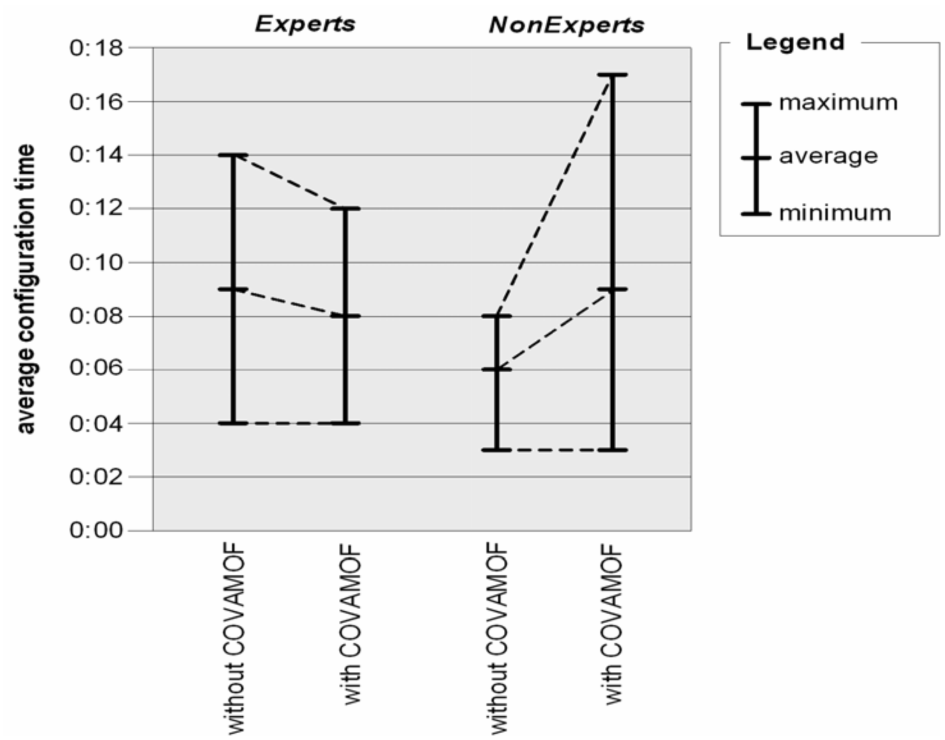


Figure 58. The effect of COVAMOF on the configuration time. This figure shows a decrease of configuration time for Experts, but an increase of configuration time for NonExperts.

The increase of configuration time for *NonExperts* raises the question for which value of the ConfigTimeFactor COVAMOF actually is still beneficial. First off all, we see an average configuration time of about eight minutes in Figure 58, which is a lot shorter than the time required to test a product. In these situations, the reduction of iterations has a larger impact on the total time than the change in configuration time.

But what would happen if we would have measured a different value for the ConfigTimeFactor? To illustrate the impact of this factor, we show its influence on the total time in Figure 61. For Figure 61, we modified Equation 1 for configuration with COVAMOF, so that it incorporates the ConfigTimeFactor:

$$total\_time_{original} = \#iterations_{original} \times (config\_time_{original} + testing\_time) \qquad (3)$$

$$total\_time_{COVAMOF} = \#iterations_{COVAMOF} \times (ConfigTimeFactor \times config\_time_{original} + testing\_time) \qquad (4)$$

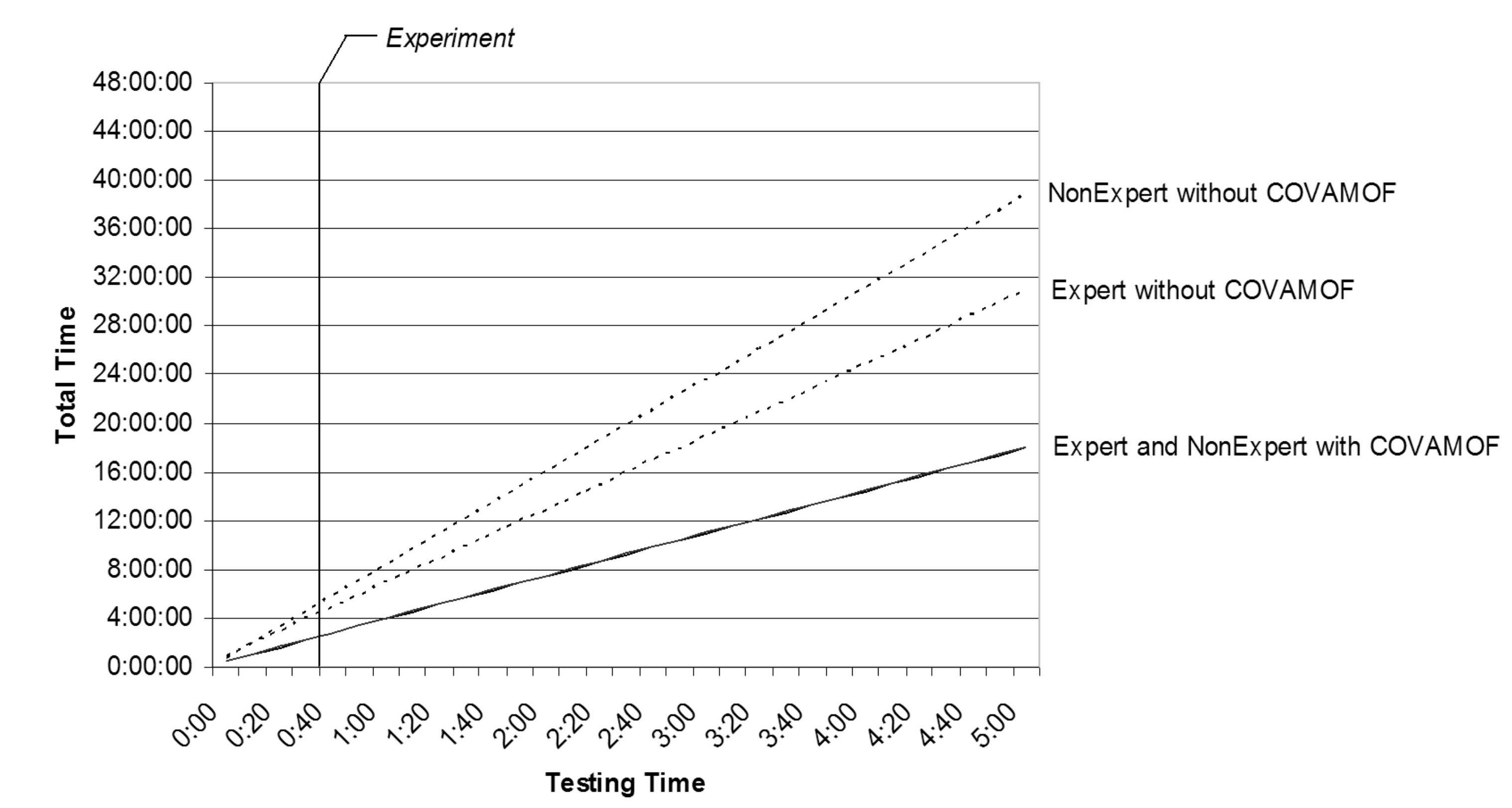


Figure 59. Total time as function of the testing time. Although the improvement factor on the total time only changes marginally for different testing times, the absolute improvement caused by COVAMOF is significant.

Figure 61 shows the total time for Experts and NonExperts as a function of the ConfigTimeFactor, where the ConfigTimeFactor ranges from 0.1 to 2.0. The vertical line in the graphs indicates the ConfigTimeFactors we measured in the experiment, i.e. $8/9 \approx 0.89$ for the *Experts* and $9/6 = 1.5$ for the *NonExperts* (see also Figure 58).

As you can see in Figure 60 and Figure 61, even when COVAMOF would have doubled the average configuration time (i.e. a ConfigTimeFactor of 2.0), its introduction still reduces the total time that would have been required to derive a product in the experiment. We can also calculate the minimum ConfigTimeFactor at which COVAMOF does not bring a reduction of this total time anymore. For *Experts*, this factor is higher or equal than 4.9 and for *NonExperts* this factor is higher or equal to 10.0.

Note, however, that instead of a ConfigTimeFactor of 4.9 for *Experts*, we actually measured a factor of 0.89 in our experiment. For *NonExperts*, we measured a ConfigTimeFactor of 1.5 instead of the 10.0 that would be required to eliminate the benefit of the reduction in iterations. We can therefore conclude that even if the effect on the configuration time would have been a lot worse than measured in the experiment, COVAMOF would still reduce the total time.

This does not mean configuration time is never important. During the tests, engineers can work on other tasks instead of waiting for the test to be finished, so a reduction of configuration time can also have its benefits. However, the effect of the configuration time is only visible if the testing time is almost equal or smaller than the configuration time, or the reduction in number of iterations is low (see also Equation 1). In most cases, the fact that *NonExperts* are now able to derive products, and the number of iterations and total time are reduced, weighs much heavier than the small increase in configuration time per iteration.

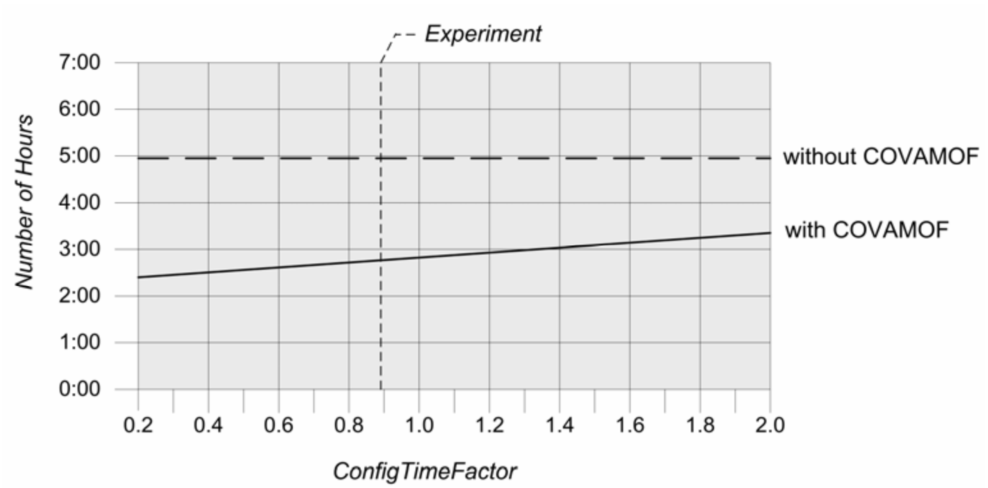


Figure 60. The impact of the ConfigTimeFactor on the total time required by Experts. As this graph shows, even an increase of the configuration time by a factor 2 does not eliminate the benefits of the reduction in number of iterations (see also Equations 3 and 4).

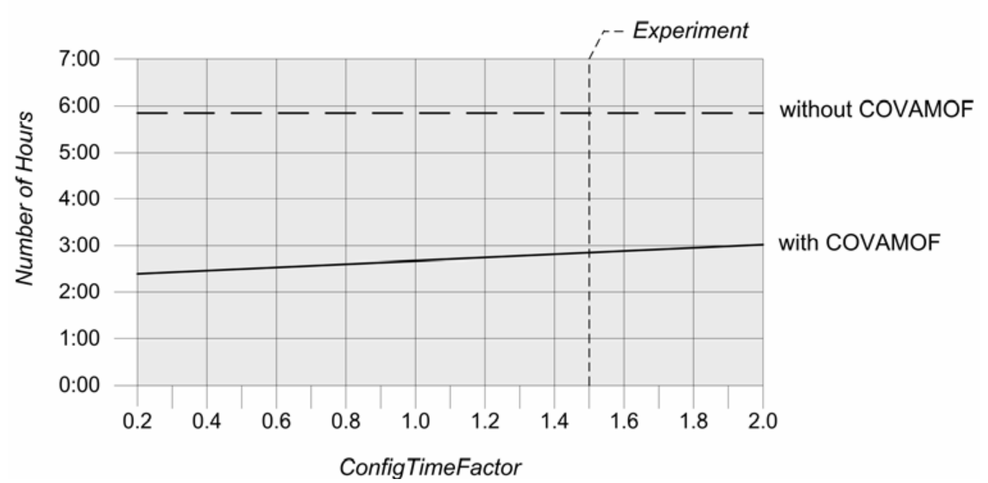


Figure 61. The impact of the ConfigTimeFactor on the total time required by non-experts. As this graph shows, even an increase of the configuration time by a factor 2 does not eliminate the benefits of the reduction in number of iterations (see also Equations 3 and 4).

10.6.5. Summary

The results of the experiment is summarized in Table 16. This table presents the average number of hours (Avg. #hours), the average number of iterations (Avg. #iterations), and the average configuration time (Avg. config time) required by experts and non-experts for product derivation with and without COVAMOF. It furthermore summarizes the percentage of participants that was able to derive products within seven iterations (%Finished). For each of these aspects we calculated the improvement from the product derivation without COVAMOF to the product derivation with COVAMOF. This improvement of a value is calculated as the difference between the value without and with COVAMOF, divided by the value without COVAMOF. To be able to calculate the reduction in iterations and total time, we counted the situations in which participants were not able to derive a product in seven iterations as requiring eight. This is the theoretical minimum they would require if they would have been allowed to continue.

So, how does this relate to our hypotheses? Hypothesis 1 states that experts should not perform worse when they use COVAMOF, and that on average, the effort required to derive products is less when using COVAMOF. As shown in Table 2 the use of COVAMOF caused the number of iterations required by experts to drop by 42%. The experiment therefore supports Hypothesis 1.

Hypothesis 2 states that with a COVAMOF model, the product derivation process is less dependent on experts, viz. non-experts will also be able to derive products, and will require less effort. As shown in Table 16, non-experts were capable of deriving products in 100% of the cases with COVAMOF, compared to 29% of the cases without COVAMOF. The effort required by the non-experts also dropped by 54%. The experiments thus also support Hypothesis 2.

10.7. Remarks from Participants

In addition to the results and observations, we received remarks from the participants. These remarks bring forward personal experiences, opinions, and ideas for improvement. In this section, we present the remarks that were made by most participants:

- “Although the experiment description listed COVAMOF-VS as a research tool, we would classify it as having the quality of a commercial product. First, the seamless integration with Microsoft Visual allowed us to quickly get familiar with the user interface and start configuring product right away. This level of maturity also brought forward confidence in the hints suggested by the COVAMOF derivation assistant.”.
- “Despite the fact that the experiment was about product derivation, we think COVAMOF can improve our actual software artifacts. We expect that during the development of these artifacts, COVAMOF will also support us in thinking about why and how we should integrate new variability. Moreover, as COVAMOF will be part of our development environment, it will also help us in documenting the right information, i.e. derivation knowledge required to derive products later on.”.

Table 16. Summary of the experiment results. The results clearly support the hypotheses that the participants would be able to derive products, and in less iterations when they used COVAMOF.

	Experts			**NonExperts**		
	without COVAMOF	**with COVAMOF**	**Improvement**	**without COVAMOF**	**with COVAMOF**	**Improvement**
% Finished	67%	100%	49%	29%	100%	245%
Average number of iterations	6	3,5	42%	7,6	3,5	54%
Average number of hours	4:32	2:49	38%	5:42	2:56	49%
Avgerage configuration time	0:09	0:08	11%	0:06	0:09	-50%

- “In the cases that COVAMOF contained directional information on variation points, suggestions given by COVAMOF only indicated whether we should decrease or increase the value of those variation points. As an improvement COVAMOF could also give more hints on how much choices should be changed. An example of such a choice is a value in the range of 0–255 with a default of 30, where after one iteration COVAMOF only suggested an increase of this default value.”.
- ” The experience we had with the method and tool has been very positive. As the FC product family involves our most complex module, we are confident that similar results will be achieved on our other modules. Dacolian B.V. will therefore soon scale up the scope of this variability model to incorporate the other modules.”

10.8. Conclusion

Despite the benefits of product family engineering, many organizations experience that product derivation is too dependent on experts, and too many iterations are required before a product is finished. In Chapter 6, we discussed why other variability modeling techniques do not sufficiently address these challenges, which was why we developed our variability management framework COVAMOF during the ConIPF project (Configuration of Industrial Product Families, Hotz et al., 2006).

The experiment we described in this chapter was started to validate whether we succeeded in achieving this goal. While experiments in the context of the ConIPF project already showed the benefits of partially formalizing variability information, we wanted to test the entire scope of COVAMOF at an organization that was not part of the ConIPF project.

In this chapter, we described the context of the experiment at Dacolian BV. Dacolian is world’s leading supplier of OEM software modules for intelligent traffic systems that utilize automatic license plate recognition (ALPR). As Dacolian faces the same problems as we identified in our case studies (see Chapter 5), they wanted to know whether COVAMOF would solve their product derivation problems. We furthermore described the hypotheses, assumptions, experiment set-up, and results of our experiment. Amongst others, we have shown why results are comparable, why we limited the maximum number of iterations to eight, how we addressed learning, and why the improvement we present in this chapter could actually be higher than we claim.

The results of the experiment showed that COVAMOF addresses the main challenges in product derivation. That COVAMOF addresses the high dependency on experts, was shown by the fact that it enabled non-experts to derive products in

100% of the cases, compared to 29% without using COVAMOF. That COVAMOF addresses the high number of iterations was shown by the fact that the effort of experts was reduced by 42%, and the effort by non-experts by 54%. These results are consistent with the results of experiments performed during the ConIPF project.

While COVAMOF was developed to address challenges that go beyond product derivation alone, this experiment forms a significant step in showing the benefits of variability modeling in general, and COVAMOF in particular. Nonetheless, several aspects remain that need to be validated. For example, with this experiment we have shown that creating a COVAMOF model and subsequently deriving products is profitable, provided that the product family does not evolve. Whether applying COVAMOF is profitable in the long term is part of subsequent experiments.

10.9. Acknowledgements

We thank all participants for their valuable time and effort. In particular, we thank Dacolian B.V. for the support they showed in successfully setting up the experiment

10.10. References

Asikainen, T., Soininen, T., Männistö , T., 2004. A Koala-based approach for modelling and deploying configurable software product families. Fifth Workshop on Product Family Engineering (PFE-5). In: Lecture Notes on Computer Science, vol. 3014. Springer Verlag, pp. 225–249.

Becker, M., 2003. Towards a general model of variability in product families. In: Proceedings of the First Workshop on Software Variability Management, Groningen, Netherlands.

Bosch, J. 2000. Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach. Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7.

Clements, P., Northrop, L. 2001. Software Product Lines: Practices and Patterns. SEI Series in Software Engineering, Addison-Wesley, ISBN: 0-201-70332-7.

Czarnecki, K., Helsen, S., Eisenecker, U., 2005. Formalizing cardinality-based feature models and their specialization. Software Process Improvement and Practice 10 (1), 7–29.

Dacolian B.V. website: http://www.dacolian.com.

Hotz, L., Krebs, T., Wolter, K., Nijhuis, J., Deelstra, S., Sinnema, M., MacGregor, J. 2006. Configuration in Industrial Product Families - The ConIPF Methodology. IOS Press, ISBN 1-58603-641-6.

v.d. Linden, F. 2002. Software Product Families in Europe: The Esaps & Café Projects. IEEE Software, Vol. 19, No. 4: 41--49.

Nonaka, I., Takeuchi, H. 1995. The Knowledge-Creating Company: How Japanese companies create the dynasties of innovation. Oxford University Press, New York.

Spillman, W.J., Lang, E. 1924. The Law of Diminishing Returns, New York, World Book Company.

Weiss, D. M., Lai, C.T.R. 1999. Software Product-Line Engineering: A Family Based Software Development Process. Addison - Wesley, ISBN 0-201-694387.

PART IV. Evolution

The last part of our variability management framework is directed towards evolution of product families. As we noted in the Introduction to this thesis, an alternative approach to decreasing application engineering cost is to make sure there are less mismatches between the variability provided by a product family and the variability required by the products. This Part presents the background, contents, and experiences of applying the COVAMOF Variability Assessment Method (COSVAM). COSVAM is the first technique for assessing variability with respect to the needs of a set of product scenarios. The five steps of COSVAM (identify assessment goal, specify provided variability, specify required variability, evaluate variability, interpret assessment results) form a structured technique that can be tuned to address a variety of situations where the question of whether, how and when to evolve variability is applicable.

Chapter 11 Variability Assessment

An important aspect of software variability management is the evolution of variability in response to changing markets, business needs, and advances in technology. In Chapter 5, we discussed that the evolution of variability should make sure it prevents mismatches between variability provided by the product family, and the variability required by the products. Variability assessment is a technique that addresses this aspect. This chapter explains what variability assessment is, and what the issues are in the current practice.

Based on	**Section numbers**
S. Deelstra, M. Sinnema, J. Nijhuis, J. Bosch, COSVAM: A Technique for Assessing Software Variability in Software Product Families, Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004), pp. 458-462, September 2004.	None, superseded by article below
S. Deelstra, M. Sinnema, J. Bosch, Variability Assessment in Software Product Families, Journal of Information, and Software Technology, conditionally accepted, 2007	All sections in this chapter, except the conclusion. The conclusion was added to link to the next chapter

11.1. Introduction

Before we go into the issues of variability assessment, the first question we need to answer is: why is variability assessment necessary? To answer this question, we go back to the work of Lehman on software evolution. He noted that as the world around us continually changes, the resulting change in purpose and context may render software products useless. It was therefore that Lehman formulated the following law on software evolution: "*A useful software system must undergo continual and timely change or it risks losing market share*" (Lehman et al., 1997). This law applies to all products in a product family.

Although variability in the product family architecture and components anticipates some of the changes in space (different products) and time (different versions of products), not all future changes can be predicted or included in the product family. Consequently, once the product family is in place, at some point in the lifecycle, evolution will force the product family to handle new functionality and thus previously discarded or unforeseen differences. In the same way that products need to undergo continual change, variability therefore has to undergo continual and timely change as well, or a product family will risk losing the ability to effectively exploit the similarities of its members.

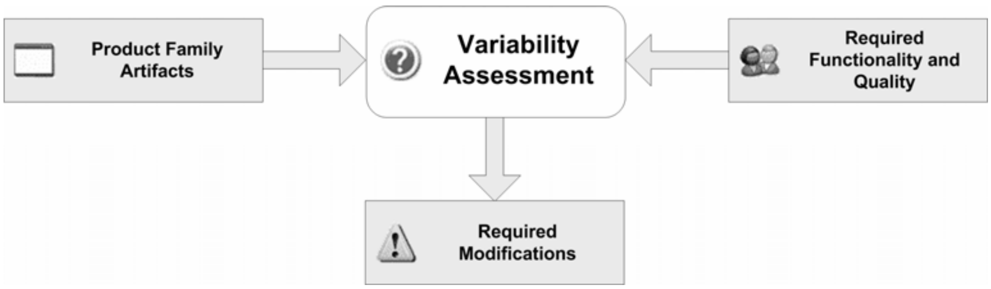


Figure 62. Variability assessment. To determine whether, how, and when variability should evolve, variability assessment evaluates whether the variability in the product family artifacts, matches the variability in the required functionality and quality.

The key challenge in this context is: “in what way can we determine whether, how, and when variability should evolve?”. A technique that deals with answering this question is what we refer to as variability assessment. Such a technique answers the question above by analyzing the mismatch between (1) the variability in the product family artifacts and (2) the variability that is demanded as necessary by the differences in functionality and quality in a set of product scenarios (see Figure 62). We call the first type of variability, *provided variability*, and the second type, *required variability*. We call the mismatch between them a *variability mismatch*.

That variability assessment is indeed relevant becomes clear from examining five common activities for software product families in which the ‘whether-how-when’-question appears:

Determine the ability of the product family to support a new product: The decision to add a new product to the portfolio depends on how well the new product fits into the product family scope. This fit depends on to which extent the required combinations of features in the new product are supported by the provided variability of the product family.

During product derivation, determine whether mismatches should be implemented in product specific artifacts or integrated in the product family: As product families are focused around a reuse infrastructure, changes can be applied product specifically, or to the reusable product family artifacts. Solving a variability mismatch by changing the reusable product family artifacts is beneficial if the short term and potentially more expensive investment in comparison to product specific adaptation, is outweighed by a decrease in cost of development effort in other products (e.g. due to a decrease in the number and severity of variability mismatches in the future). Solving a variability mismatch in the product family furthermore depends on whether it is possible or desirable to apply changes to the product family artifacts. A solution may, for example, change dependencies in such a way that dependency values for existing products cannot be met anymore. Variability assessment in this context therefore involves assessing which combinations of features of a new product are not supported by the provided variability, and where potential mismatches need to be solved.

Collecting input data for release planning: During the lifecycle of a product family, organizations collect characteristics (such as functionality and quality) that are required and desired for new and existing products. These characteristics are retrieved from, for example, customers, market analysis and technology forecasting. The result of this collection is typically a long list of required and desired characteristics per product. Due to organizational, economical and technical constraints, however, not all of these characteristics can be implemented in the next release of the products or reuse infrastructure. Release planning is therefore concerned with deciding when to release different versions of the products, including the selection of characteristics offered in specific versions as well as decisions concerning the inclusion of these characteristics in the reuse infrastructure. Release planning involves balancing the objectives regarding the organizational (e.g. staff restrictions), economical (e.g. cost and revenue), and technical (e.g. technical feasibility of feature combinations) constraints. The accuracy of the answer to this problem is, amongst others, influenced by the accuracy of the effort estimates for integrating characteristics product specifically and in the product family, as well as the accuracy of determining in which combinations characteristics can be integrated in the product family artifacts. Variability assessment in this context thus involves identifying mismatches as a result of a set of new product releases, as well as determining how mismatches should be solved.

Assess the impact of new features that cross-cut the existing product portfolio. Some organizations develop a product portfolio that consists of a set of products that interact with each other (e.g. an organization providing systems at both server- and client-side). Rather than focusing on adding entire products or product versions, the focus of variability assessment in this context is on assessing the impact of adding one or more features that influence multiple products in the product family.

Determine whether all provided variability is still necessary. Variation points and variants become obsolete when the need to support different alternatives disappears during evolution, or when predictions made during proactive evolution turn out to be incorrect. In Chapter 5, we identified that the existence and lack of removing obsolete variability was one of the underlying causes of complexity, and had a detrimental effect on the efficiency of product derivation. The aim of assessment in this context is to identify provided variability that is obsolete with respect to the variability required by products that have been or will be developed, and to determine how the product family should respond.

As software product families in industry have been widely adopted and evolve constantly, organizations that employ product families already perform some form of variability assessment. In the two sections below, we discuss the issues with current approaches (both in practice and related work).

11.2. Variability Assessment Issues

Before a new product is derived, for example, a specification of the product functionality and quality is handed to, typically, software architects. The task of these architects is to assess how much effort will be associated in delivering the product with this specific set of functionality and quality. From what we have seen in several case studies that our group has participated over the years (e.g. as described in Chapter 3-5, or the Dacolian case described in the Chapter 4), current approaches that are used to determine whether, when, and how variability should evolve, are associated to a number of *methodological* and *knowledge* issues.

The *methodological* issues refer to the problems associated to the principles and procedures of current approaches.

Unstructured: Variability assessment is often done by architects without explicit methodological guidance. Instead, they employ an informal process based on their own common sense and experience. These informal processes are typically highly unpredictable with respect to their outcome and required effort.

Reactive instead of proactive: Assessments are furthermore often only applied in case of immediate problems or needs. As a consequence, these assessments suffer from time-pressure and lack of availability of experts; both for the assessment process, and for applying solutions.

Generalized instead of optimal decisions: A third issue is that, in some cases, decisions with respect to evolving variability are generalized over a number of features. In Chapter 4, we presented some extreme forms of generalization we found in industry. For example, one business unit would apply all necessary changes product specifically for each release of the product family, while another business unit would incorporate all necessary changes in the reusable product family artifacts. Both cases lead to problems. Where in the first case the full reuse potential of the product family is not utilized (they re-implement similar functionality in each single product), the second case leads to an unnecessary increase of complexity.

Lack of removing obsolete variability: After a while, the purpose of certain variation points and variants may disappear. Functionality specific to some products can become part of the core functionality of all product family members (Bosch et al., 2001), or perceived alternatives may not be needed after all. Assessments, however, usually only consider the necessity and feasibility of including new functionality, but do not evaluate existing variability with respect to its actual use. The result is an abundance of obsolete variation points and variants. They lead to a situation in which the complexity of the product family only increases during evolution, and the predictability and traceability only decreases. In addition, obsolete variation points result in a situation where engineers start to forget about the provided variability. During the case studies we described in Chapter 5, for example, the interviewees indicated the existence of obsolete parameters from which no one knew what they are for, let alone what the optimal value was.

Addressing only one layer of abstraction: Most existing assessment techniques only focus on one layer of abstraction, i.e. either the architecture (e.g. Clements et al. (2001), Folmer et al. (2004)), or its implementation in code (e.g. Bohner (2001) and Kung et al. (1994)). However, variability is a concern that crosscuts all layers of abstraction. In case a detailed list of changes to the product family artifacts (both architecture and components) is required, this issue therefore drives the need for a technique that is able to address all these layers in a uniform fashion.

The *knowledge* issues refer to the problems associated to the information on which decisions in an assessment are based.

Implicit variability: The last methodological issue (addressing only one layer of abstraction) suggests using a variability model that relates variability information across different abstraction layers. In many organizations, however, no complete and explicit model is available that covers all these layers. As the time and effort that is available for an assessment is limited, specifying a complete explicit model is often not an option. Not using a model at all also proofs problematic, however, as it is difficult to keep an overview of all variation points and their relations (see Chapter 5).

Neglecting implementation dependencies: Even if particular options for functionality and quality are independent from a problem space perspective, the design and implementation of a product family can create additional dependencies between them. The consequence of dependencies as a result of implementation is twofold. First, not all combinations of options provided by a product family can be offered in one product without modification. This means that even if all required options are provided by the product family, the required combination of options may not be available. Second, effort estimates for new functionality and quality cannot be considered independent from other changes, as those may also have implementation dependencies.

Insufficient number of alternative solutions: When a variability mismatch occurs, several solution strategies may exist to address this mismatch. For example, the software architect has to decide whether to solve a mismatch product specifically, in the reuse infrastructure, or not at all. In addition, he or she can choose from different mechanisms to solve the mismatch. The choice for a particular solution depends on a trade-off between pros and cons of the potential solutions. Examples of pros and cons of a solution are: whether it introduces incompatibilities in the asset base due to new dependencies, whether it imposes the use of immature or unstable technology, and how it affects the effort associated with other changes. The issue we address here involves software architects that only consider a very small number of alternatives, rather than carefully looking for the optimal solution (Bosch et al., 2001).

These knowledge issues cause assessments to produce non-optimal and inaccurate results. The consequence is that, during product derivation, unexpected incompatibilities are identified. Chapter 5 explains that these incompatibilities have a profound impact on the total effort and time-to-market for the product at hand.

11.3. Related work

Existing techniques have been suggested for variability assessment. In the discussion on related work below, we relate variability assessment to existing work on product families, and discuss why existing approaches are not suited to address all variability assessment issues we discussed above.

FAST and SEI's Product Line Practice. Weiss and Lai (1999) formulate basic assumptions with respect to product families, from which two are particularly important in the context of this article: "it is possible to predict the changes that are likely to be needed to a system over its lifecycle", and "it is possible to take advantage of these predicted changes". When it comes to actually determining changes to variability, however, the book lacks precision (p. 198): "… You can adapt standard change management techniques to FAST projects, so the FAST PASTA model does not elaborate on those aspects in any great detail". Also in the SEI's Product Line Practices and Patterns book, the evaluation of variability is quoted as an example of an important evaluation. The book, however, does not present a technique to perform these evaluations. Rather, it suggests modifying existing architecture assessment methods to accomplish this goal (pp. 77-83).

Investment analysis. James Whitey (1996) provides an investment analysis approach that focuses on maximizing ROI of product line assets. Robertson and

Ulrich (1998) also evaluate economical aspects of a product family and deal with planning and scoping the product family architecture. DeBaud and Schmid (2003) provide a similar but more general approach. They also claim that product-centric commonality and variability analysis is better than a domain based view, as the latter provides a flawed economic model for making scoping decisions (DeBaud and Schmid, 2003). The approaches, however, are all based on rough estimates, and focus on the question if certain features, assets or products should be part of the product family rather than how required variability should be realized in the product family artifacts.

Assessment. Assessments typically consist of five steps, i.e. goal specification, specification of the provided aspect, specification of the required aspect, analyzing the difference between the provided and required aspect, and interpreting the results. Examples of these approaches are ATAM (Clements et al., 2001), ALMA (Bengtsson et al., 2004), and SALUTA (Folmer et al., 2004). These three approaches respectively assess trade-offs between quality attributes, maintainability, and usability in software architectures. The approaches differ with respect to the required information, elicitation and specification of the scenarios. They focus on analyzing the architecture of single systems, rather than being suitable for all layers of abstraction in a product family.

An approach that does address variability, is the approach presented by Wijnstra (2003). The approach presents a high-level discussion on extracting variability information, and, based on this information, evaluates the provided variability with respect to best practices. However, it is restricted to end-user variability, does not provide specific steps for building up a provided variability specification, and is not focused on specific variability needs of the product family members.

Change management. Change management is a process for ensuring that changes to a software system or product family are traceable, carefully planned, and motivated. The change management process is typically a high-level description of how a change should be handled, defines standard deliverables, as well as an organizational structure. Variability assessment neatly fits into the change management process in product families. Where the descriptions in change management process usually do not go further then saying there are steps such as 'propose change', or 'evaluate change', variability assessment provides the details that are required to actually propose and evaluate changes to the variability.

11.4. Conclusion

The variability assessment issues we identified above, prevent giving a good answer to the question whether, when, and how variability should evolve. As a response, we have developed the COVAMOF Software Variability Assessment Method (COSVAM). We present this method in the next chapter.

11.5. References

Bengtsson, P.O., Lassing, N., Bosch, J., van Vliet, H., 2004. “Architecture-level Modifiability Analysis (ALMA)”, Journal of Systems and Software, Vol. 69(1-2), pp. 129-147.

Bohner, S.A., 1991. Software Change Impact Analysis for Design Evolution, In Proceedings of 8th International Conference on Maintenance and Re-engineering (ICMR’91), Los Alamitos, California, pp. 292-301.

Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, H., Pohl, K., 2001. Variability Issues in Software Product Lines, Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), pp. 11–19.

Clements, P., Kazman, R., Klein, M., 2001, Evaluating Software Architectures, Methods and Case Studies, Addison-Wesley, ISBN 0-201-70482-X.

DeBaud, J.M., Schmid, K., 1999. “A systematic approach to derive the scope of software product lines”, Proceedings of the 21st Int. Conf. on Software Engineering, California, USA, pp. 34-43.

Folmer, E., Gurp, J., Bosch, J., 2004, “Architecture-Level Usability Assessment”, accepted for EHCI.

Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., and Chen, C., 1994. Change Impact Identification in Object Oriented Software Maintenance, Proceedings of the International Conference on Software Maintenance (ICSM’94), IEEE CS Press, Los Alamitos, California, pp. 202-211.

Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., Turski, W.M., 1997. "Metrics and Laws of Software Evolution - The Nineties View", Proceedings of the Fourth International Software Metrics Symposium, USA.

Robertson, D. Ulrich, K., 1998, “Planning for product platforms”, Sloan Mgt. Review, Vol. 39 (4), pp. 19-31.

Weiss, D.M., Lai, C.T.R., 1999, Software Product-Line Engineering: A Family Based Software Development Process, Addison-Wesley, ISBN 0-201-694387.

Whitey, J., 1996. “Investment analysis of software assets for product lines”, Technical report CMU/SEI-96-TR-010, Software Engineering Institute.

Wijnstra, J.G., 2003. Evolving a Product Family in a Changing Context, Proceedings of the 5th International Workshop on Software Product-Family Engineering (PFE-5), pp. 111-128.

Chapter 12 COSVAM

To be able to determine whether, when, and how variability should evolve, we have developed the COVAMOF Software Variability Assessment Method (COSVAM). The contribution of COSVAM is that it prescribes a structured assessment process that addresses the issues that are associated to the current variability assessment practice. In this chapter, we present an elaborate description of COSVAM and discuss the experiences of applying COSVAM in an industrial setting.

Based on	**Section numbers**
S. Deelstra, M. Sinnema, J. Nijhuis, J. Bosch, COSVAM: A Technique for Assessing Software Variability in Software Product Families, Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004), pp. 458-462, September 2004.	None, superseded by article below
S. Deelstra, M. Sinnema, J. Bosch, Variability Assessment in Software Product Families, Journal of Information, and Software Technology, conditionally accepted, 2006	All sections in this chapter

12.1. Overview

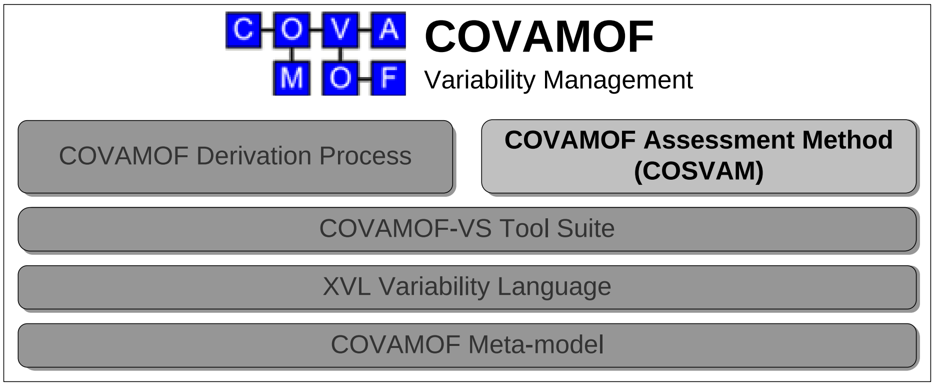


Figure 63. COVAMOF. COVAMOF is a variability management framework that is currently built-up from five main parts, including COSVAM.

The COVAMOF Software Variability Assessment Method (COSVAM) was developed as part of our variability management framework COVAMOF (see Figure 63). The idea behind COSVAM is visualized in Figure 64. It shows the main deliverables that are produced during COSVAM, i.e. the COVAMOF Provided Variability Model, COVAMOF Required Variability Model, Mismatches, Solution Scenarios and Required Modifications:

- **COVAMOF Provided Variability Model:** This is a COVAMOF model of the variability in the product family artifacts (see also Chapter 8).
- **COVAMOF Required Variability Model:** This is also a COVAMOF model. However, this model specifies the variability that is required by the functionality and quality required for a number of product scenarios. In COSVAM, this model is specified in terms of configurations of the COVAMOF Provided Variability Model.
- **Mismatches:** The Mismatches are the differences between the provided and required variability, such as new variation points, conflicts between variants, and unmet quality attribute values. As the COVAMOF Required Variability Model is specified in terms of the COVAMOF Provided Variability Model, the Mismatches are based on new entities and conflicts in the COVAMOF Required Variability Model, rather than a comparison between the two models.
- **Solution Scenarios:** The Solution Scenarios specify possible ways to solve the variability Mismatches. Each Solution Scenario has a different impact on the product family.
- **Required Modifications:** Required Modifications specify a set of changes to the product family artifacts that optimally address the needs and constraints of the product family. This set is based on a comparison of the pros and cons of the Solution Scenarios.

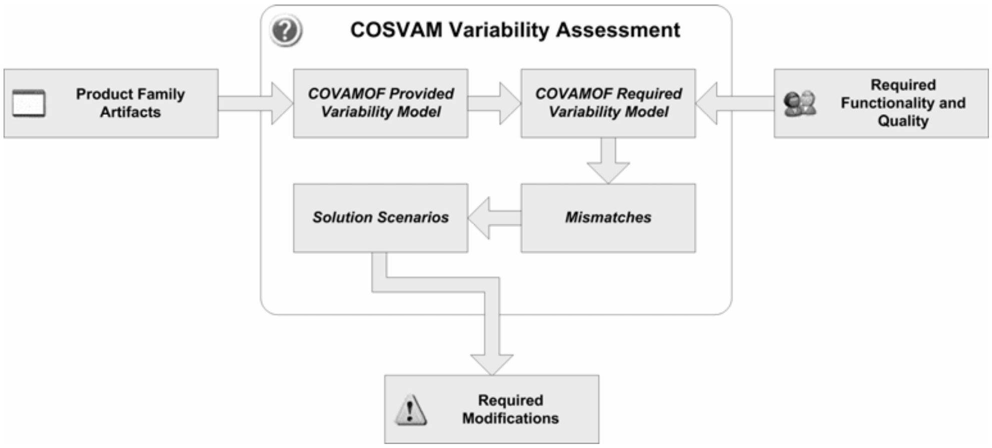

COSVAM Variability Assessment
Product Family Artifacts
COVAMOF Provided Variability Model
COVAMOF Required Variability Model
Required Functionality and Quality
Solution Scenarios
Mismatches
Required Modifications

Figure 64. The structure of COSVAM. This figure denotes the main deliverables of the assessment method when the purpose is to gather input for release planning.

To produce these deliverables, we defined an iterative process. This process divides the assessment problem into five Steps (see Figure 65). COSVAM aims for manageability, and repeatability, and therefore distinguishes multiple goals, explicitly identifies the sources of information, and forces assumptions and decisions to be made explicit. Each Step describes how results should be obtained, processed, and interpreted:

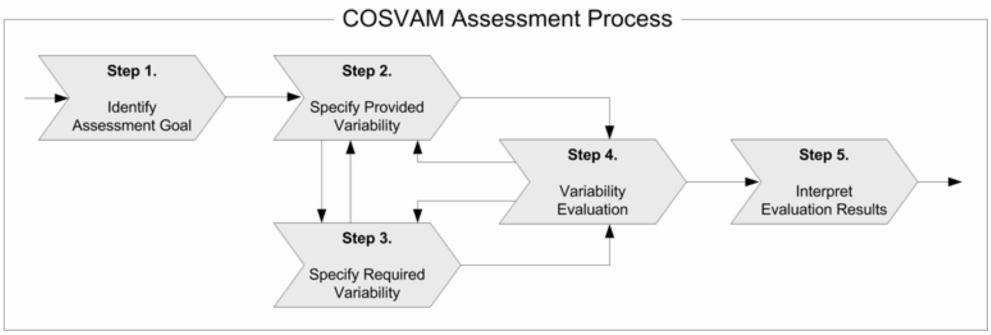

COSVAM Assessment Process
Step 1.
Identify Assessment Goal
Step 2.
Specify Provided Variability
Step 3.
Specify Required Variability
Step 4.
Variability Evaluation
Step 5.
Interpret Evaluation Results

Figure 65. The COSVAM Assessment Process. This assessment process breaks down in five Steps, numbered from 1 to 5. Although this numbering identifies the default sequence in the Steps, iterations can occur between Step 2, 3 and 4.

Step 1 and 5. As COSVAM is designed to be used in different situations, and with different goals in mind, the method starts with clearly demarcating the assessment goal, and ends with an interpretation step. The first Step, i.e. identify assessment goal, identifies the assessment context, required outcome, scope, and required team members, and plans the remaining Steps. The last Step, i.e. interpretation, interprets the evaluation outcome with respect to the assessment goal. These Steps allow optimally structuring this process, viz. to tune the input, output and activities of each Step to a particular situation. By forcing assumptions and decisions to be made explicit, COSVAM furthermore allows proposing changes to the provided variability that optimally address the needs and constraints, where the rationale behind these changes can be traced.

Step 2 and 3. The second and third Step involve specifying the provided and the required variability, respectively. As in many product family organizations variability is implicit, the purpose of the second Step in COSVAM is to specify the provided variability uniformly over several abstraction layers. As the time for an assessment is typically limited, COSVAM addresses the problem of not being able to create a complete specification by focusing on a particular scope (identified in the first Step), and iterating between specifying provided and required variability.

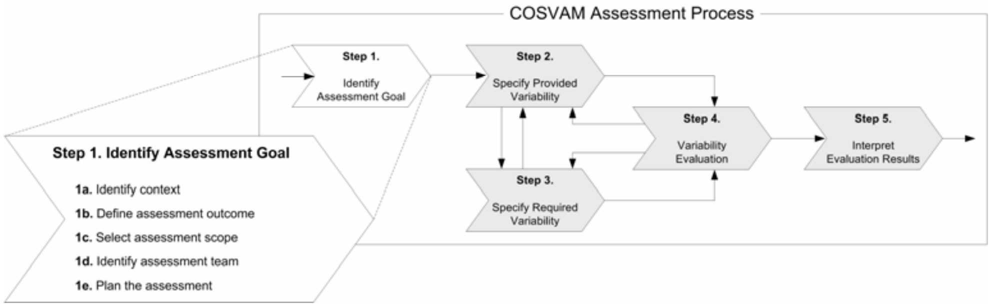
The purpose of the third Step is to specify the variability that is required to accommodate the combinations of functionality and quality in the set of product family members that are within the assessment scope (i.e. product scenarios). Depending on the goal of the assessment, this Step also existing products as scenarios to be able to identify obsolete variability, and future products to be able to proactive evolve the product family.

Both Steps use a COVAMOF model to specify the types of variability. To prevent a complicated comparison between models that potentially have a different decomposition and naming of entities, the COSVAM specifies the required variability in terms of the configuration of existing entities in the provided variability model, as well as differences to existing entities in that model. As a part of the required variability typically matches the provided variability, this also saves modeling effort in specifying required variability.

Step 4. The fourth Step, i.e. variability evaluation, involves finding out how well the required variability is supported by the provided variability, and what changes can be made to accommodate mismatches between them. These mismatches are clustered, and for each cluster several alternative solutions are designed (i.e. solution scenarios). For each solution scenario the impact is determined in terms of, for example, required effort, as well as impact on design, time-to-market, tooling, performance, and testing.

Each Step of COSVAM consists of a number of Activities. In the following sections, we discuss these Activities in detail. We describe how each Activity is performed, and how results are delivered. We illustrate each Activity with our case study at Dacolian B.V. (see Chapter 4). The focus of our discussion is on the situation where COSVAM is used to collect input data for release planning.

12.2. Step 1 Identify Assessment Goal



COSVAM Assessment Process
Step 1. Identify Assessment Goal
Step 2. Specify Provided Variability
Step 3. Specify Required Variability
Step 4. Variability Evaluation
Step 5. Interpret Evaluation Results
Step 1. Identify Assessment Goal
1a. Identify context
1b. Define assessment outcome
1c. Select assessment scope
1d. Identify assessment team
1e. Plan the assessment

Figure 66. Identify Assessment Goal. This first Step of the COSVAM assessment process breaks down into four Activities, numbered from 1a to 1d.

Software variability assessment can pursue different goals, under very different circumstances. Both have an impact on how the assessment should be performed. The purpose of the first Step of COSVAM is therefore to identify the goal and circumstances under which the assessment is performed. Making these goals and circumstances explicit ensures all members of the assessment team face the same direction. It also allows tuning the assessment input, output, and process, so that the assessment optimally suits the goal, and addresses the circumstances.

The output of this Step is an Assessment Goal Document, which is maintained and used throughout the whole COSVAM assessment. It contains a description of the goal, circumstances, their influences, and organization of the assessment. To produce the Assessment Goal Document, COSVAM defines four Activities in this Step, i.e. Identify context (Activity 1a), Define assessment outcome, (Activity 1b), Select assessment scope (Activity 1c), and Identify assessment team (Activity 1d). We summarize these Activities in Figure 66 and discuss them below.

12.2.1. Activity 1a - Identify context

We associate the context of the assessment to why and how organizations initiate the assessment. When we motivated the relevance of a variability assessment technique in the Introduction, we discussed five situations in which variability assessment plays a role, i.e. (1) *Determine the ability of the product family to support a new product*, (2) *Determine whether mismatches should be implemented in product specific artifacts or integrated in the product family, (3) Collecting input data for release planning, (4) Assess the impact of new features that cross-cut the existing product portfolio, (5) Determine whether all provided variability is still necessary.*

Each of these situations describes *why* a particular assessment is initiated. A second aspect of the assessment context is *how* the assessment is initiated. An assessment is typically initiated in three different ways:

- A *periodic* assessment is regularly initiated, e.g. once per month, or once per year. Examples include assessment for release planning, and benchmarking the organization.
- An *event-driven* assessment is marked by a significant event. Examples include the start of deriving a new product version, and extending the product portfolio.
- A *stakeholder-initiated* assessment is initiated by one or more influential members of the organization. An example is a reverse assessment, i.e. where the assessment team determines whether the provided variability actually matches the variability required by products that already have been built. Such an assessment can be initiated after members of the organization complain about the product family artifacts.

The reason why the assessment was initiated has an impact on the required input and output of the assessment. Determining the ability of the product family to support a new product, for example, only requires considering the variability required as a result of one product, while the other situations require a particular set of products. How the assessment is initiated, on the other hand, has a considerable impact on the cooperation, time pressure, and acceptance of the assessment results afterwards. In a periodic assessment, for example, the organization is typically committed to the assessment, while in an event-driven or stakeholder-initiated assessment not all members of the organization may see the need, and the process potentially suffers from time pressure, political forces, and lack of commitment.

Due to the impact of the context on the assessment, identifying the assessment goal starts with identifying this context. The result of this Activity is a description of the reason and way of initiating the assessment, as well as an overview of factors that may influence the assessment.

Table 17. Output of Step 1, Activity 1a: Identify context.

Output Name	**Representation**	**Usage**
Situation	A description of why the assessment was initiated	Step 1, 2, 3, 4 and 5
Initiation	*periodic, event-driven* or *stakeholder-initiated*	Step 1, 2, 3, 4 and 5
Influencing Factors	Table in the in the Assessment Goal Document that contains the *names* and *descriptions* of the influencing factors.	Step 1, 2, 3, 4 and 5

Dacolian Case Study: *COSVAM was initiated at Dacolian to plan the new release of the Intrada ALPR product family, i.e. version 5.0.0 (see also Chapter 4). In the first years of the Intrada ALPR product family, releases used to be stakeholder-initiated. An important customer would ask for new functionality, and the only question an assessment would need to handle was: "how much time and effort is associated to the change". As the number of customers increased, and the organization and product family matured, however, the focus of release planning changed from determining the time and effort required for a change to a situation in which an optimal set of changes is determined. Now, major releases are planned every year at Dacolian, so the assessment initiation is periodic. The most important influencing factor involved the maximum amount of effort available for the assessment.*

Table 18. Output of Activity 1a in the Dacolian Case Study

Output Name	***Value***	
Initiation	*Periodic*	
Influencing Factors	***Name***	***Description***
	Effort	*Development for a release currently starts 5 months prior to actual delivery, and the design and development constraints for changes necessary for the new release are approximately 20 person-months. The assessment itself was initiated five weeks prior to the start of development.*

12.2.2. Activity 1b - Define assessment outcome

The second Activity in this Step is defining the outcome of the assessment, while taking into account the Influencing Factors from Activity 1a. This identification breaks down into three different aspects. The first aspect of the assessment outcome is the results that are directly related to the variability of a software product family. Examples of these results include a list of specific mismatches between provided and required variability, an overview of possible solutions for these variability mismatches, effort estimates, an overview of decisions with respect to selecting optimal solutions, or a qualitative measurement of the suitability of the provided variability in the product family.

These results influence the extent in which the COSVAM process is executed. When a list of mismatches is required, for example, the assessment team can skip the parts in which different solutions are considered from the fourth Step (Variability Evaluation). The second aspect is therefore the impact of the expected results on the assessment process.

The third aspect of the assessment outcome is related to communicating the results. The assessment team has to take care of how the results of the assessment are presented to the organization, depending on the context of the assessment. In case of unfamiliarity with COSVAM, for example, additional supportive evidence with respect to the reliability of the assessment may be required. People may furthermore be offended, or feel threatened in their position in case of negative results. Next to a precise definition of the assessment results and the impact on different parts of the assessment process, the definition of the assessment outcome therefore also consists of a description of how the results will be communicated to the organization.

Table 19. Output of Step 1, Activity 1b: Define assessment outcome

Output Name	Representation	Usage
Expected Results	Section in the in the Assessment Goal Document that describes what type of results have to be produced.	Activity 1c, 5a
Impact on Process	Section in the in the Assessment Goal Document that describes which process Steps and Activities should be performed to produce the results.	Step 1, 2, 3, 4 and 5
Communication	Section in the in the Assessment Goal Document that describes how the results can be optimally communicated to the organization.	Activity 5c

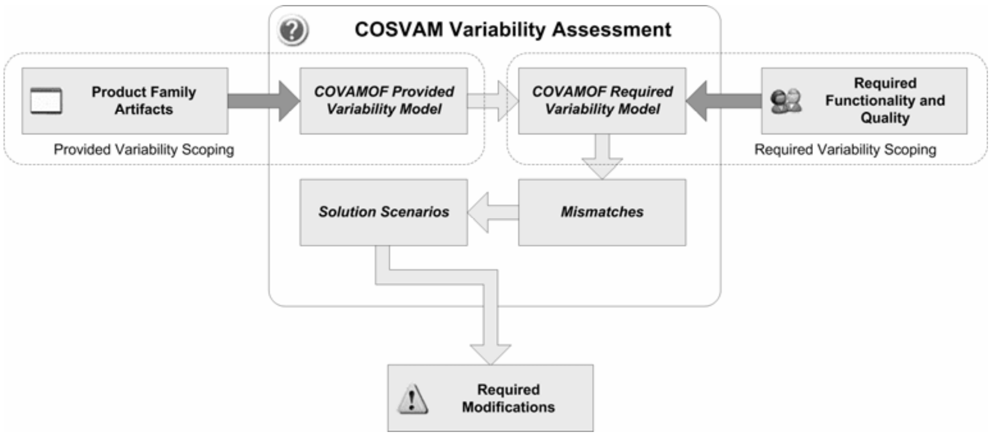
Dacolian Case Study: *At Dacolian B.V., this Activity resulted in the following output:*

Table 20. Output of Activity 1b in the Dacolian Case Study

Output Name	***Value***
Expected Results	*Our aim of assessment for release planning is twofold. The first aim of the assessment is to be able to communicate to customers, as early as possible, which variability will be included in the product family. The second aim is to communicate to the organization, which changes we have to make to the product family artifacts. The assessment outcome will therefore be separated into external and internal results. The external results are an overview of new features that will be offered by the new release. The internal results specify which changes will be applied to the variability provided by the product family, how changes depend on each other, the impact on tooling for testing and validation, an overview of affected modules per change, and effort estimates. As our aim is to maintain a configurable product family that does not require product specific adaptations, the internal results will only specify which changes we should apply to the product family, and which changes we will not offer.*
Impact on Process	*This particular assessment outcome requires the whole COSVAM process to be executed. The types of results expected from this assessment are based on the results that are always expected from our release plans."*
Communication	*Using the assessment, a small team decides which changes will be incorporated in the next release of the Intrada ALPR product family. These decisions are communicated to the other engineers at Dacolian B.V. in the form of an MS PowerPoint document that is presented in a meeting. The external results are communicated in the standard external release document, which specifies the API changes in the product family.*

12.2.3. Activity 1c - Select assessment scope

The third Activity in identifying the assessment goal is defining where the assessment will focus on. A complete assessment involves the entire product family, all existing and future products, product versions, all their functionality and quality, the entire product family architecture, and all reusable components. As time and effort is limited, however, the assessment team typically has to limit the required and provided variability scope.



COSVAM Variability Assessment
Product Family Artifacts
COVAMOF Provided Variability Model
COVAMOF Required Variability Model
Required Functionality and Quality
Provided Variability Scoping
Required Variability Scoping
Solution Scenarios
Mismatches
Required Modifications

Figure 67. Scoping Provided and Required Variability. The Provided Variability Scope (dashed box on the left) determines which parts of the product family will be considered when creating the COVAMOF Provided Variability Model (dark arrow on the left). The Required Variability Scope (dashed box on the right) determines how the size of the product scenarios will be limited when creating the COVAMOF Required Variability Model (dark arrow on the right).

The *required variability scope* states how the size of the product scenario set will be limited when specifying required variability in Step 3:

- *Time Scope:* This scope deals with the timeframe from which products releases will be considered during the assessment. The assessment can be focused on future products that will use the next release of the product family, future products that span across different releases, or only present-day products. In addition, old products can be used in a retrospective study to identify whether all variability was really necessary.
- *Space Scope:* This scope deals with the products that will be considered during the assessment. The assessment is typically limited to one product if the purpose of the assessment is to identify how well that product is supported by the product family, and which product specific changes should be made to accommodate mismatches. This is not suitable for identifying what should be changed in the product family, as other product family members could pose restrictions on the necessary changes. For identifying product family changes, at least a subset of products should be used. This subset could, for example, be the products that generate most revenue.
- *Functionality and Quality Scope:* Rather than focusing on all product functionality and quality, the level of detail of the required variability can be limited further by focusing on the key features of products in the product scenario set. This scope specifies the targeted level of detail that will be considered during the assessment.

The *provided variability scope* states which parts of product family will be considered when specifying provided variability in Step 2. The assessment team can limit the scope of the assessment to the provided variability of a subset of the abstraction layers, or a selection of subsystems:

- *Abstraction Layer Scope:* this limits the assessment to one or more abstraction layers in the product family. For example, when limiting the assessment to the feature and architecture layer, many implementation details can be left implicit, and the assessment primarily depends on the involvement of architects.
- *Subsystem Scope:* this limits the assessment to a selection of subsystem of the product family, thus requiring only a portion of the component experts or application engineers in externalizing the provided variability.

The selection of the scope depends on the Expected Results defined in Activity 1b. The result of limiting the provided and required variability scope is twofold. First, the amount of information that has to be externalized and managed considerably decreases. Second, fewer experts have to be involved in the assessment, thus limiting the intervention with everyday business. Basically, limiting the assessment scope limits the time and effort required for the assessment by focusing only on extracting information that is required to produce the Expected Results. However, it also restricts the level of detail of the assessment, which may influence the accuracy of results. Limiting the assessment scope thus also introduces risks; the assessment process may require additional iterations in case of missing information, and the results may introduce problems if they are incorrect.

To ensure traceability of these decisions, the result of this Activity therefore not only consists of a description of which provided and required variability scope was selected, but also the rationale, impact and risks of the selected scope. The rationale specifies why a particular scope was chosen, the impact specifies the influence of the decision on the assessment process and results, and identifies points that have to be taken into consideration when using this slice, while the risks specify potential problems as a result of the selected scope.

Table 21. Output of Step 1, Activity 1c: Select assessment scope

Output Name	Representation	Usage
Time Scope	Section in the Assessment Goal Document that describes the time window from which product releases are considered in the assessment. This text furthermore specifies the *rationale*, *impact* and *risks* of this decision.	Step 1, 3 and 4
Space Scope	Section in the Assessment Goal Document that describes the different products that are considered in the assessment. This text furthermore specifies the *rationale*, *impact* and *risks* of this decision.	Step 1, 3 and 4
Functionality and Quality Scope	Section in the Assessment Goal Document that gives an overview on the targeted level of detail in the required functionality and quality This text furthermore specifies the *rationale*, *impact* and *risks* of this decision.	Step 1, 3 and 4
Abstraction Layer Scope	Section in the Assessment Goal Document that discusses the level of detail of the assessment in terms of abstraction layers, together with the *rationale*, *impact* and *risks* of this decision.	Step 1, 2 and 4
Subsystem Scope	Section in the Assessment Goal Document that specifies on which subsystems the assessment will be focused, together with the *rationale*, *impact* and *risks* of this decision.	Step 1, 2 and 4

Dacolian Case Study: *The output of this Activity at Dacolian is described in Table 22. For the Time and Space Scopes, see also Figure 16.*

Table 22. Output of Activity 1c in the Dacolian Case Study

Output Name	***Value***
Time Scope	*The assessment window will span 15 months. This means we will take into account the products that we expect to sell in the one and a half year. We will also consider the old products during the assessment. However, as they are only used to predict which customer might buy new products in the next year, we consider them as input for specifying the required variability. We don't look ahead further than fifteen months, as we feel we are not able to predict accurately over a longer time span due to the rapid changes in our domain.*
Space Scope	*We will focus our assessment on products in the Tolling, and Law Enforcement market segments, and not on the Access Control segment. The reason for selecting this scope is twofold. First, we are the best and largest player in the Tolling, and Law-Enforcement segments, while the Access Control segment is highly competitive, with many players. Second, the functionality that is demanded for Access Control is totally different from the other segments, and not in our core competence. The revenue for Access Control is less than 5% of the total revenue, so the impact and risk of not incorporating Access Control is also small. Only when we have to choose between alternative changes to the product family, and these changes score equally, we will look at the impact of the change on Access Control products.*
Functionality and Quality Scope	*All functionality and quality in our domain is relevant, so we do not restrict the scope upfront.*
Abstraction Layer Scope	*We do not restrict the level of detail of the assessment to a particular abstraction layer. This will require externalization effort for specifying provided variability in all layers. However, as COSVAM also allows limiting the externalization effort to only the information that is required in Step 2, we choose to save effort there. That way, we do not to limit the accuracy of the assessment upfront by scoping to a particular abstraction layer.*
Subsystem Scope	*We will not look at variability in the ALPR Licensing component. This component determines whether the customer has a valid license for the ALPR product. Changes to this module are difficult, and have many ripple effects. In our experience, the required variability scope will also not trigger necessary changes to the licensing component. Excluding this component saves us externalization effort, and does not introduce risks for the assessment results.*

12.2.4. Activity 1d - Identify assessment team

This Activity involves selecting the participants that are required to be able to deliver the desired assessment outcome from Activity 1b, within the context and selected provided and required variability scopes from Activity 1a and 1c, respectively. For example, a variety of experts are needed as the assessment involves a situation with a large amount of implicit information, predicting the future, and estimating effort. To let the assessment run smoothly, it should be clear for each participant what his or her own role is, as well as the role of the other participants.

The result of this Activity is a list of experts and their role in the assessment. Roles that should at least be fulfilled in the assessment are:

- *Chairman:* The chairman is responsible for running the assessment. He or she takes care that each Step of the assessment is carefully carried out, organizes the meetings, is responsible for staying on schedule, and guides discussions.
- *Scribe:* The scribe captures the results of Activities, and the discussions that led to those results.
- *Provided variability expert:* The provided variability experts are required to externalize the provided variability in Step 2. Examples of these experts are software architects to identify architectural variability, product developers that know existing product configurations, and have experience with the provided variability of the product family, or component experts
- *Required variability expert:* The required variability experts are participants that are able to provide information that is necessary to create product scenarios in Step 3. Examples of these are technology experts, innovators, and marketing managers.
- *Solution expert:* Once mismatches between required and provided variability are identified solutions have to be proposed in Step 4. This requires the participations of, for example, architects and developers.
- *Business expert:* To be able to relate different solutions to the business goals and constraints in Step 5, the assessment also needs business experts. An example of a business expert is a project manager that is able to provide productivity figures.

Note that the first two roles, i.e. Chairman and Scribe are not specific to COSVAM. They also appear in other assessment methods. ATAM (Clements et al., 2001), for example, specializes the role of chairman and scribe into the roles Team leader, Evaluation leader, Scenario scribe, Proceedings scribe, Timekeeper, Process observer, Process enforcer and Questioner. The other roles depict the type of persons that are necessary for specific Steps of COSVAM.

Table 23. Output of Step 1, Activity 1d: Identify assessment team

Output Name	**Representation**	**Usage**
Roles	List of role-names in the Assessment Goal Document.	Steps 1,2,3,4,5
Participants	Table in the Assessment Goal Document that contains the participants' *names* together with their *roles* (i.e. the name from the Roles-table specified above) and the *set of Steps* in which they are involved.	Steps 1,2,3,4,5

Dacolian Case Study: *The output of this Activity at Dacolian is presented in Table 24.*

Table 24. Output of Activity 1d in the Dacolian Case Study

Output Name	***Value***		
Roles	*Component Expert, Sales Manager*		
Participants	***Role***	***Name(s) and Function***	***Steps***
	Chairman	*Architect 1*	*Step 2, 3, 4, and 5*
	Scribe	*Architect 1*	*Step 2, 3, 4, and 5*
	Provided variability expert	*Architect 1, Product developer 1, Architect from the Intrada Systems product family.*	*Step 2*
	Required variability expert	*Sales Manager 1, 2, and 3*	*Step 3*
	Solution expert	*Architect 1 and 2*	*Step 4*
	Business expert	*Project Manager 1*	*Step 5*

12.2.5. Activity 1e – Plan the assessment

The combined results from the Activities above form the goal of the assessment and are specified in the Assessment Goal Document. Once this goal has been specified, it is important to plan the assessment, obtain commitment from the organization for both the assessment and use of assessment results, and to assemble the assessment team. For a more detailed discussion on these aspects see e.g. Clements et al. (2001), and Maccari (2002).

Table 25. Output of Step 1, Activity 1e: Plan the assessment

Output Name	**Representation**	**Usage**
Assessment Schedule	A document that describes how much effort will be spent on each Step, as well as the dates and times of the assessment meetings.	Steps 2,3,4,5

Dacolian Case Study: *The effort for each of the remaining Steps of the assessment was roughly divided as follows. The specification of provided and required variability was estimated to take a maximum of one week. The analysis of mismatches in the evaluation Step would be supported by tooling. The analysis was estimated to take anywhere from 2 hours to 4 days per configuration, depending on the mismatches that would be identified during the assessment. The automated analysis would allow Dacolian to run multiple analyses in parallel. Any analysis that would likely take more than 3 days would be simplified to a quicker, but less accurate derivative analysis, or expert judgment. The design and impact analysis of the solutions was estimated to take a maximum of three weeks. One additional day was estimated for the interpretation Step. As a result, each of the Steps would fit in the five weeks set for the assessment.*

The commitment for the assessment was available from at least two senior managers. A meeting with respect to obtaining commitment for applying the necessary changes from the developers was postponed to after the assessment results would be ready.

12.3. Step 2 - Specify Provided Variability

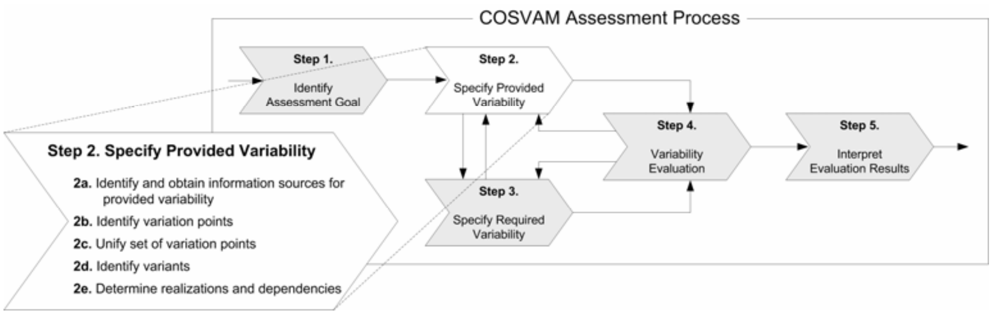

COSVAM Assessment Process
Step 1.
Identify Assessment Goal
Step 2.
Specify Provided Variability
Step 4.
Variability Evaluation
Step 5.
Interpret Evaluation Results
Step 3.
Specify Required Variability
Step 2. Specify Provided Variability
2a. Identify and obtain information sources for provided variability
2b. Identify variation points
2c. Unify set of variation points
2d. Identify variants
2e. Determine realizations and dependencies

Figure 68. Specify Provided Variability. In this second Step of the COSVAM, the COVAMOF provided variability model is incrementally specified during five Activities, numbered from 1a to 1e.

In many organizations, variability is implicit, or specified in different types of models that have no explicit relations across abstraction layers. The purpose of this Step is therefore to specify the provided variability of a product family in a variability model where variability is linked across abstraction layers. COSVAM uses a COVAMOF model to create such a specification (see the Introduction for a brief discussion on the main entities in COVAMOF).

As the time and effort for an assessment is typically limited, it is usually impossible to create a COVAMOF model that completely represents all variability provided by a product family. In COSVAM, this problem is addressed in two ways. First, in Activity 1c, the provided variability scope has been limited through the Abstraction Layer and Subsystem Scopes. The provided variability model will only contain variability that is within these two scopes.

The second way in which the COSVAM addresses the balance between the large amount of implicit information and a limited amount of time and effort, is by continually weighing how much information will be externalized throughout this Step, and how much will left implicit. Stopping criteria for Activities in this Step are when the experts in the assessment team are confident that externalizing more information will not significantly impact the assessment results. They can use the Functionality and Quality Scope for the required variability (defined in Activity 1c) to determine these stopping criteria. This Step furthermore interacts with Step 3 (Specify Required Variability) and Step 4 (Variability Evaluation) to find out which artifacts should be involved in the comparison between provided and required variability (see also Figure 68).

The team documents any decisions to partially externalize certain variation points or complete parts of the product family. When during a later stage in COSVAM it turns out that information contained in these parts is necessary, the assessment team can decide to iterate this specification Step (see Figure 65), or leave the information implicit. The decisions to leave parts implicit are used in the interpretation Step of the assessment (Step 5).

The result of this Step is thus a COVAMOF model that partially covers the provided variability of a product family. This specification models variability in terms of a COVAMOF model, where variation points and dependencies are first-class and related across lifecycle phases. To produce such a model, the provided variability specification Step is structured in five Activities: Identify and obtain information sources for provided variability (Activity 2a), Identify variation points (Activity 2b), Unify set of variation points (Activity 2c), Identify variants (Activity 2d), and Determine realization rules and dependencies (Activity 2e). Figure 68 summarizes these Activities.

12.3.1. Activity 2a - Identify and obtain information sources for provided variability

The specification Step starts with identifying information sources that can be used to model the variability provided by the product family artifacts. Potential information sources include: expert knowledge, feature diagrams, the product family architecture, component documentation, product configurations, make-files, and source code. In case this Activity identifies that a provided variability model is already available in the right form and detail, the other Activities can be skipped completely, and the assessment can proceed with the next Step of the assessment, i.e. specifying required variability.

Table 26. Output of Step 2. Activity 2a: Identify and obtain information sources for provided variability

Output Name	Representation	Usage
Provided Variability Sources	Table that contains for each source the *name*, a *description* and the *location* of the source.	Activity 2b,c,d,e

Dacolian Case Study: *At the time of the assessment there was no COVAMOF variability model of all their ALPR product family artifacts yet. They therefore developed a new model during this Step, for which they identified a total of ten different types of input. A part of the output of this Activity is shown in Table 27:*

Table 27. Sample of the output of Activity 2a in the Dacolian Case Study

Output Name	***Value***		
Provided Variability Sources	***Name***	***Description***	***Location***
	C-interface specifications	*Each component is associated with an interface-file. This file specifies the different ways in which a component can be used, and was therefore a first source of component variability.*	*Intrada ALPR Source tree*
	Component source-files	*As not every variable aspect of a component is visible to the outside, the component source-files were a second source of component variability.*	*Intrada ALPR Source tree*
	Internal module documentation	*The internal module documentation specifies instruction to employees of Dacolian that need to compile the module before it is delivered to the customer. This documentation specifies the rationale behind different manual steps that can be taken during compilation.*	*File Cabinet with the ALPR documentation*
	…	…	…
	Tooling	*Part of our product derivation process is supported by proprietary tooling. Many of the post-deployment settings are done through these tools. In addition, the Intrada ALPR product family contains MDA-based code generation tools that can generate a variety of source files.*	*Dedicated Network Drive*
	License file	*Our Intrada ALPR products contain a license mechanism. The license files of Intrada ALPR product family members specify restrictions on which variants a customer is allowed to use. From these files we can read which product configuration was produced for this customer.*	*Scattered over drives of different participants of our assessment*

12.3.2. Activity 2b - Identify variation points

Once the Provided Variability Sources are obtained in Activity 2a, the next Activity is to use these sources to identify the first of the two main aspects of variability, i.e. the variation points. These variation points identify locations in product family artifacts that are within the Provided Variability Scope, and that provide the ability to vary functionality and quality between different products. A number of approaches can be used find these variation points:

- Some existing notations, such as for feature diagrams, already specify variability. The Assessment Team can identify variation points directly from these notations.
- Interview Provided Variability Experts, and let them draw from experience to identify important variation points.
- Compare the architecture, selected components, and parameter settings in different product configurations to identify differences.

- Search for patterns in design and implementation that may indicate variability. These patterns consist of design patterns (Gamma et al., 1995) such as factories and abstract classes, but also code statements such as if-statements and pre-compiler directives.
- Search for comments that deal with choices in design and code. Although specifying criteria for an automated search with full-coverage may prove complicated, comments can be identified by experts, and often serve as valuable addition to the other means for externalizing variability above.

Table 28. Output of Step 2. Activity 2b: Identify variation points

Output Name	**Representation**	**Usage**
Identified Variation Points	Table that contains, for each identified variation point, a *description*, its *location* and the *source* (from the Provided Variability Sources table in Activity 2a).	Activity 2c

Dacolian Case Study: *The experts were asked to identify variability in features and draw different architectures. They used a search for code statements such as pre-compiler, if-, and switch-statements in the source files and c-interface specifications to identify variation points in the source code. The internal module documentation, component make-files and partial component configurations were used to identify variation in the component structure. The reference manuals for customers and tooling were used to identify post-deployment variability. The MDA-based tooling was furthermore inspected to identify variability with respect to the generated source code. This Activity identified a total of 6872 variation points.*

Table 29. A sample from the output of Activity 2b in the Dacolian Case Study

Output Name	***Value***			
Identified Variation Points	***Name***	***Description***	***Location***	***Source***
	DS_OS_XXXX	*#ifdef that selects the operating system (Win32, WinCE, Linux, Solaris)*	*Throughout the source files*	*.h and .c files in the Source Tree*
	Countries	*The countries where our ALPR products is able to recognize license plates from*	*Download section*	*Web server*
	…	…	…	…

12.3.3. Activity 2c - Unify set of variation points

The identification of variation points results in a set of variation points that originate from different sources. The purpose of unification is to identify multiple representations of the same variation point, to relate variation points with Realization Relations, to remove too low-level variation points, and to determine their attributes. The unification process furthermore identifies settings that are identical across multiple product configurations.

In this Activity, assessment team members develop the initial COVAMOF variability model from the Identified Variation Points (Activity 2b) and additional information from the Provided Variability Sources (Activity 2a). For each unique identified variation point in the Identified Variation Points they create a new Variation Point entity in this model, and specify, where known, its attributes. The most important attributes are the *name*, the *description*, the *references* to the associated rows in the "Identified Variation Points" table, the *type*, the *abstraction layer*, the *binding time*, whether it's *opened* or *closed*, and its *realization technique*. Similar, for each identified realization, they create a new COVAMOF Realization Relation entity in the variability model, which is linked to the Variation Points (see Chapter 8).

Table 30. Output of Step 2. Activity 2c: Unify set of variation points

Output Name	Representation	Usage
Initial Provided Variability Model	COVAMOF Model with Variation Points and Realization Relations.	Activity 2d

Dacolian Case Study: *During the previous Activity, the assessment team made sure no multiple representations from the same variation point were identified. Most of the variation points identified during the previous Activity (94.6%) were value variation points that were identified in two components (called Neural-Network and Matcher) that are generated by MDA-based tools, and commented variables in the other source files. The assessment team decided not to externalize all information regarding these component implementations, as the variability provided these components was considered to be well isolated from other variability. In addition, it was expected that the variability provided by these components would be enough to address any changes in the ALPR domain during the Time Scope. This reduced to total amount of value variation points to be explicitly considered during the assessment to approximately 1%. The remaining variation points were almost equally distributed among the other types (1.7% optional, 1.2% alternative, 1% optional variant, and 1.5% variant). Most of the value variation points had their binding time and closing time at compilation time. For a sample of the resulting COVAMOF model of Step 2, see Figure 69.*

12.3.4. Activity 2d - Identify variants

The unified set of Variation Points in the Initial Variability Model is then populated with Variants. A number of Variants have already been encountered earlier, as these Variants served to identify Variation Points in the first place. This set of Variants is not always complete however, as only a subset of product configurations may have been used, or because the product configurations only use a subset of the available Variants. The set of Variants can therefore be extended by using additional information from the Provided Variability Sources, such as interviewing experts, inspecting additional product configurations, and analyzing the set of shared product family assets. This process is guided by the Variation Points in the COVAMOF model.

Table 31. Output of Step 2, Activity 2d: Identify variants

Output Name	Representation	Usage
Extended Provided Variability Model	COVAMOF Variability Model with Variation Points, Realization Relations, and Variants.	Activity 2e

Dacolian Case Study: *The previous Step primarily yielded closed Variation Points with the latest binding time at post-deployment. As a consequence all Variants of these Variation Points were specified in the source code, and identified during the previous Activities. For the compile-time Variation Points there are only a limited amount of Variants as well, so all of those Variants had also been identified in the previous Activities. For a sample of the resulting COVAMOF model of Step 2, see Figure 69.*

12.3.5. Activity 2e - Determine realization rules and dependencies

After the unification of the set of Variation Points, the Realization Relations specify which Variation Points are related to each other. Up until this Activity, however, three things are missing. First, the Realization Relations only specify *that* Variation Points are related to each other, not yet *how*. Second, the variability model does not specify which combinations of Variants are permitted, and third, it does not specify how the selection of Variants affects system properties such as quality attributes (see also Chapter 8). This Activity therefore deals with adding realization rules and Dependencies to the provided variability model. After this Activity, the COVAMOF provided variability model is ready to be used in subsequent Steps.

Complementing the Realization Relations with realization rules is important, as they specify how the selection of one or more Variants at one Variation Point *realizes* to the selection of Variant at other Variation Points. Specifying Dependencies is necessary to get an accurate assessment results. Dependencies that denote restrictions between Variation Points are important as they restrict the possible choices in the product family. Without these Dependencies, mismatches are potentially not identified, as the product family would seem to provide more variability than actually available. Dependencies that denote quality attributes are important as the assessment would otherwise only focus on functional aspects of the product family. Dependencies are also indispensable for determining necessary changes and their impact.

The assessment team determines the realization rules and Dependencies from the Provided Variability Sources (see Activity 2a). For example, the differences between product configurations provide clues as to which configuration settings lead to different combinations of features and thus are used to determine realization rules. Other tools, such as static code analysis, inspection, and interviews are also used to either determine how Variation Points realize each other, or to determine Dependencies that denote which Variants require or exclude each other.

Modeling Dependencies that specify system properties is typically more complicated. First, the assessment team has to identify the important system properties, e.g. by interviewing experts and inspecting requirements specifications. For each of these properties, they create a Dependency entity in the Variability Model. Second, they have to identify *which* Variation Points influence the value of the system property. For each Variation Points that influences a Dependency, they create a new Association entity for that Dependency. Third, the assessment team specifies, where known, *how* the Variation Points influence the system properties of the associated Dependencies.

Besides interviewing experts and studying documentation, there are specific techniques that can be used to determine these influences. A fairly simple technique is to modify the settings for choices in existing product configurations, and to test these modified configurations. A technique that takes this approach a step further, and that uses test results from *existing* product configurations is called sensitivity analysis. This approach originates from Taguchi's robust design method (Taguchi and Phadke, 1984). In Taguchi's method it is used to determine the influence of single parameters, from a set of data points where multiple parameters were changed between measurements. We can translate this to the problem in product families. The values of the system properties of existing product configurations are our measurements. These product configurations thus define data points, where multiple choices were changed between measurements.

Finally, the assessment team enriches the knowledge on Dependencies in the variability model with Dependency Interactions, typically by using derivation knowledge of product family experts. Note that specifying the Dependencies is a time-consuming task. Here, the assessment team thus also has to weigh how detailed the analysis of the Dependencies should be. In Chapter 8, we describe how COVAMOF allows the team to limit the level of detail of the knowledge on the influence of Variation Points on system properties.

Table 32. Output of Step 2. Activity 2e: Determine realization rules and dependencies

Output Name	Representation	Usage
Provided Variability Model	COVAMOF Model with Variation Points, Realization Relations, Variants, and Dependencies.	Step 3 and 4

Dacolian Case Study: *During the unification of the set of Variation Points, the assessment team related the Variation Points with Realization Relations. They did not address the realization rules explicitly in this Activity, however. The main reason for not explicitly specifying realization rules was that the experts indicated that the variability in the Intrada ALPR product family was organized in such a fashion that choices in one layer straightforwardly map to choices in another layer. According to these experts, the provided variability also hardly contained any implementation Dependencies that would cause combinations of Variants not being supported by Variation Points in lower layers. In addition, most realization rules were already formalized in the source code.*

The primary action during this Activity therefore comprised of identifying natural Dependencies between features (through interviews), and the Variation Points that influence the main quality attributes (through interviews, and product configurations). To identify how the Variation Points influence the main quality attributes, several configurations were tested during the lifecycle of the previous release. These comprised of running 100 typical configurations on 70 datasets (each containing millions of images). For a sample of the resulting COVAMOF model see Figure 69 and Figure 70. Figure 70 shows the result of incrementally building up the COVAMOF model. It features a screenshot where the COVAMOF-VS Add-ins visualize a part of the variability in the ALPR product family.

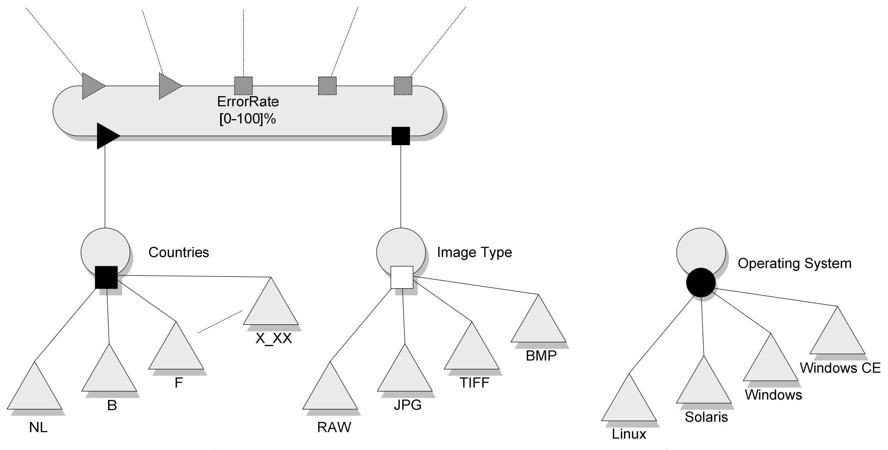
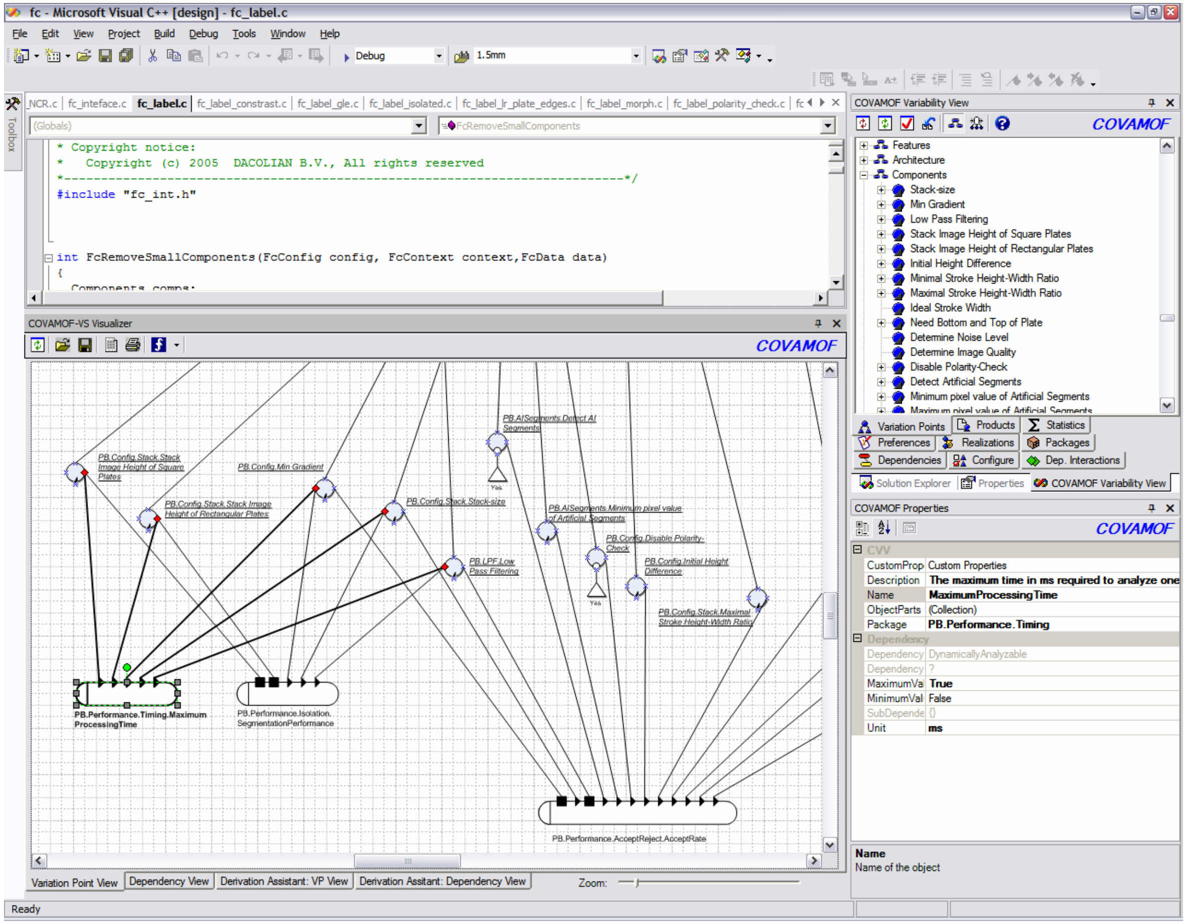


Figure 69. A sample from the Provided Variability Model from the Dacolian case study.


fc - Microsoft Visual C++ [design] - fc_label.c
File Edit View Project Build Debug Tools Window Help
Debug
1.5mm
NCR.c | fc_inteface.c | fc_label.c | fc_label_constrast.c | fc_label_gle.c | fc_label_isolated.c | fc_label_lr_plate_edges.c | fc_label_morph.c | fc_label_polarity_check.c | fc
(Globals)
FcRemoveSmallComponents
COVAMOF-VS Visualizer
COVAMOF
PB.Config.Stack.Stack Image Height of Square Plates
PB.Config.Min Gradient
PB.Config.Stack.Stack Image Height of Rectangular Plates
PB.Config.Stack.Stack-size
PB.AISegments.Detect AI Segments
PB.AISegments.Minimum pixel value of Artificial Segments
PB.Config.Disable Polarity-Check
PB.LPF.Low Pass Filtering
PB.Config.Initial Height Difference
PB.Config.Stack.Maximal Stroke Height-Width Ratio
Yes
PB.Performance.Timing.Maximum ProcessingTime
PB.Performance.Isolation.SegmentationPerformance
PB.Performance.AcceptReject.AcceptRate
Variation Point View | Dependency View | Derivation Assistant: VP View | Derivation Assitant: Dependency View
Zoom:
Ready
COVAMOF Variability View
COVAMOF
Variation Points | Products | Statistics
Preferences | Realizations | Packages
Dependencies | Configure | Dep. Interactions
Solution Explorer | Properties | COVAMOF Variability View
COVAMOF Properties
COVAMOF

```
* Copyright notice:
*   Copyright (c) 2005  DACOLIAN B.V., All rights reserved
*------------------------------------------------------------------------------*/
#include "fc_int.h"


int FcRemoveSmallComponents(FcConfig config, FcContext context,FcData data)
{
  Components comps;
```

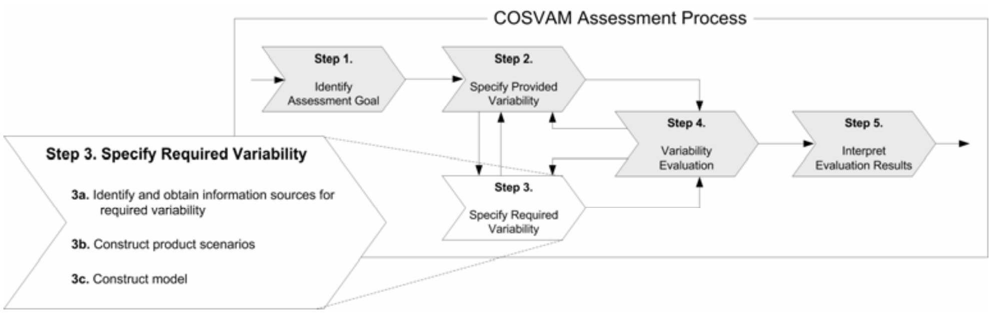
- Features
- Architecture
- Components
  - Stack-size
  - Min Gradient
  - Low Pass Filtering
  - Stack Image Height of Square Plates
  - Stack Image Height of Rectangular Plates
  - Initial Height Difference
  - Minimal Stroke Height-Width Ratio
  - Maximal Stroke Height-Width Ratio
  - Ideal Stroke Width
  - Need Bottom and Top of Plate
  - Determine Noise Level
  - Determine Image Quality
  - Disable Polarity-Check
  - Detect Artificial Segments
  - Minimum pixel value of Artificial Segments
  - Maximum pixel value of Artificial Segments

CVV	
CustomProp	Custom Properties
Description	The maximum time in ms required to analyze one
Name	MaximumProcessingTime
ObjectParts	(Collection)
Package	PB.Performance.Timing
Dependency	
Dependency	DynamicallyAnalyzable
Dependency	?
MaximumVa	True
MinimumVal	False
SubDepende	0
Unit	ms

Name
Name of the object

Figure 70. Screenshot of COVAMOF-VS. This figure shows a screenshot of a part of the provided variability model of the ALPR product family, as visualized by the COVAMOF Add-ins for Microsoft Visual Studio.NET. These Add-ins use the MS Visio Active-X control to display variability graphically.

12.4. Step 3 - Specify required variability



COSVAM Assessment Process
Step 1. Identify Assessment Goal
Step 2. Specify Provided Variability
Step 3. Specify Required Variability
Step 4. Variability Evaluation
Step 5. Interpret Evaluation Results
Step 3. Specify Required Variability
3a. Identify and obtain information sources for required variability
3b. Construct product scenarios
3c. Construct model

Figure 71. Specify Required Variability. In this third Step of the COSVAM assessment process, the COVAMOF required variability model is incrementally specified during three Activities, numbered from 3a to 3c.

The previous Step results in a model of the *provided* variability. In the third Step of COSVAM, the associated Participants (specified in Table 23) use this model to specify the *required* variability. The required variability model captures the variability that is required to accommodate the combinations of functionality and quality in the set of product family members that are within the required variability scope. This required variability scope was already defined during the first Step of COSVAM (see Activity 1c).

This Step consists of three Activities, i.e. Identify and obtain information sources for required variability (Activity 3a), Construct product scenarios (Activity 3b), and Construct model (Activity 3c). We summarize these Activities in Figure 71. The Impact on Process and Influencing Factors from Step 1, and the Provided Variability Model from Activity 2e serve as input, and the Required Variability Model is the output of this Step. In the next Step (Step 4 - Variability Evaluation), the required variability is compared to the provided variability. To prevent a complicated comparison between models that potentially have a different decomposition and naming of entities, the COSVAM specifies the required variability in terms of the configuration of existing entities in the provided variability model, as well as differences to existing entities in that model. An additional benefit of this solution is that effort is saved in modeling the required variability. For a more detailed explanation of this solution, see Activity 3c.

12.4.1. Activity 3a - Identify and obtain information sources for required variability

Specifying required variability starts with selecting the information sources that are necessary to determine the required variability. Example sources include existing and future product specifications, market analysis, different roadmaps, internal variability requirements, and a catalog of requirements requested by customers.

The selection of these sources depends on the Required Variability Scopes (see Activity 1c), as well as the availability of information. The required variability scope limits the required information sources to a particular subset of products, and a particular subset of features. The availability of information determines what information is part of the input of the assessment, and what information has to be constructed during the Activities in this Step. When future product specifications are already available, for example, they are part of the information sources. Otherwise, specifying future products is part of the Activities in this Step, and thus market analysis, customer requests, and roadmaps are necessary inputs. For more examples of sources and pointers to literature, see Clements and Northrop (2001).

Table 33. Output of Step 3. Activity 3a: Identify and obtain information sources for required variability

Output Name	Representation	Usage
Required Variability Sources	Table that contains for each source the *name*, a *description* and the *location* of the source.	Activity 3b and 3c

Dacolian Case Study: *Three sources were identified for specifying required variability, i.e. a roadmap from innovation, product descriptions of products that had been built on top of version 3.0 and 4.0 of the Intrada ALPR family (see Figure 16), and a change request catalog, that is divided into internal and external change request. For the previous release, the change request catalog contained 50 change requests. Of these 50 requests, 35 were internal request, and 12 originated from existing users. For the new release, the catalog contained 100 requests that were mapped onto 12 feature branches. The number of internal requests and requests from existing users remained constant over the previous and new release. The increase in requests is thus primarily due to the increase in visibility and reputation of the ALPR products, which yielded a substantial increase in potential customers. This also meant that a number of the request were outside of the existing domain of the Intrada ALPR product family (e.g. from the existing image recognition that involve country names, to image recognition that involve container numbers). The output of this Activity is described in Table 34.*

Table 34. Output of Activity 3a in the Dacolian Case Study

Output Name	***Value***		
	Name	***Description***	***Location***
Required Variability Sources	*Roadmap from innovation*	*The roadmap from innovation is included as it contains, for example, new technical insights from of ongoing internal research, and scientific publications.*	*Computer of System Architect 1*
	Product descriptions	*The product descriptions contained descriptions of product specific changes, and therefore would identify changes that might qualify for inclusion the in the product family.*	*File Cabinet*
	External change requests	*The external change requests originate from two sources, i.e. existing users and potential customers. Existing users are customers that have already bought OEM-products. They will buy new products during the new release, or have a maintenance contract that entitles them to a new product version. Change requests from these existing users are often more detailed than change requests from potential customers. These potential customers are customers that have asked for a product during the previous release(s), but that could not be served with the existing product family. As Dacolian strives for a configurable product family, these customers are not provided with a product that contains product specific changes, but are put on hold to the next release.*	*Change request catalog*
	Internal change requests	*The internal change requests originate from three sources, i.e. market analysis, the Intrada Systems family, and the Product Developers. Market analysis for the ALPR family analyzes the market for improvements that are required to maintain a competitive advantage. Market analysis also takes customer specific change requests to find out whether these changes have a broader applicability than a specific customer, and whether they can be generalized to different alternatives. The Intrada Systems family serves as internal customer for the ALPR family. They are separated from external customers as external requests have an economic priority over internal requests. The requests from the Product Developers involve change requests that will improve the product derivation process.* *Change requests typically involve both the feature and the component layer. As the change requests originate from different sources, they are specified in different forms. Each change request is therefore translated to a uniform ontology, and mapped to a feature branch to cluster changes before entering the catalog. The change request catalog preserves the source, the priority of the source, and frequency in order to determine whether a request is required, desires, goodwill to existing customers or will increase sale.*	*Change request catalog*

12.4.2. Activity 3b - Construct product scenarios

The next Activity is using the Required Variability Sources to construct the product scenarios. Product scenarios consist of an overview of the functionality and quality of products that exist or are perceived within the required variability scope of the assessment (i.e. Scope in Space and Time). The scenario construction Activity breaks down into the following aspects.

- The assessment team defines (or generates) the Feature Dictionary that initially contains the Variant and Dependency entities and their descriptions from the Provided Variability Model. This dictionary is maintained throughout the whole Specify Required Variability Step and ensures consistent meanings of features and quality attributes (see also domain analysis methods such as FODA (Kang et al. 1990).
- They then identify the functionality and quality attributes that are present in the domain in the timeframe specified by the required variability scope (for example, based on existing products, change requests, technology roadmaps, and market analysis). These are added to the Feature Dictionary. From this set, the assessment team identifies which functionality and quality attributes fit in the selected Functionality and Quality Scope.
- The assessment teams then identifies the product releases that are in the Space and Time Scope (i.e. the product scenarios). An overview of these scenarios is drawn in a similar fashion to the product portfolio evolution diagram in Figure 16.
- For each product identified above, the team selects a combination of functionality and quality that is likely to be needed for the product scenarios, and describes this scenario in the Product Scenario Description Document. In order to make sense during the assessment, the quality attributes must be measurable at some point in the product lifecycle. Required quality can be required range, value, or a minimum/maximum. They also identify the variation within each product scenarios: the *type* of variation, and the *earliest* and *latest* time each Variant should be bound (e.g. compile-time or runtime).

Note that depending on how much information is already available, one or more results of this construction process may already be part of the input of this Activity (see also Activity 3a).

Table 35. Output of Step 3. Activity 3b: Construct product scenarios

Output Name	Representation	Usage
Feature Dictionary	Table that contains the features and quality from the Feature Dictionary that are in the scope of the assessment.	Activity 3c
Product Scenario Overview	A diagram that presents an overview of the scenarios that exist, or are perceived, within the Time and Space Scopes.	Activities 4d, 4e, 5b, and 5c
Product Scenarios Description	For each product scenario, a section in the Product Scenarios Description document, with references to Sources and terminology from the Features Overview. Each section describes where the product that is defined by the scenario is located in Time and Space, and least contains a description of the functionality, variation, and quality in the product.	Activities 3c

Dacolian Case Study: *The number of product scenarios that would have to be created during this Activity was estimated to be approximately 500 (based on featured products, requested new products, and desired products). Rather than specifying each individual product scenario, the assessment team chose to specify a number of generic product scenarios, as well as a number of specific scenarios. The specific scenarios comprised of products configurations that were based on request from existing customers. The specific scenarios therefore consisted of the features the customers previously requested, as well as the product specific changes for these existing customers.*

Below, we present five examples of required variability that are related to the samples of provided variability in Figure 69. First, if we look at the Image Type Variation Point, Intrada ALPR should support more image file formats. One of the product scenarios requested the support for JPEG2000 and some profiles requested the support for RAW Bayer (i.e. a RAW variant with alternating blue, green and red pixels). All input images of Intrada ALPR version 4.0 furthermore contained eight bits per channel per pixel. For customers from Texas (US) and Taiwan this was not sufficient. These customers also required twelve bits and wanted to be able to switch between these types of images at runtime.

The second example of required variability in the Product Scenarios involved constraints on the recognition performance of Intrada ALPR. Where in the scenarios for Law Enforcement the recognition rate and error rate should be at least 80% and at most 1% respectively, for Tolling products, these values are 90% and 0.1%.

Third, the Product Scenarios required Intrada ALPR to support the recognition of license plates from 41 additional countries. Intrada ALPR version 4.0 supported only 16 different countries. Moreover, future Intrada ALPR products should be *capable of recognizing license plates from multiple countries at once. Although combining different countries was technically already possible in version 4.0, such combinations implied a huge drop in recognition performance, in particular at a large number of countries. More precise, in version 4.0, the number of countries in one product should not exceed three; otherwise the error rate was not acceptable anymore. Some Product Scenarios from Activity 3b, though, required more than five countries.*

Fourth, some customers did not want their products to recognize all different plates from one country, but instead wanted to limit the recognition of Intrada ALPR to one or more specific plate styles. Examples of these plate styles are license plates of standard passenger cars, motors, trucks, trailers, royal vehicles, and military vehicles, as well as temporary license plates.

Finally, Intrada ALPR 5.0 should support four new Linux distributions. Where customers of Intrada ALPR 4.0 on a Linux system all used the SUSE 7.1 distribution, in the future, customers will also run Intrada ALPR on other distributions (i.e. SUSE 9.0, Redhat ES 3.0, Fedora Core and Redhat 9.0). As Dacolian did not know whether all Intrada ALPR 4.0 libraries would work on these new distributions, the Intrada ALPR product family should explicitly provide support for these distributions.

Table 36. Samples from the output of Activity 3b in the Dacolian Case Study

Output Name	***Value***
Feature Dictionary	

Feature name	***Description***
single-image	*There is only one input image of each car that can be used to recognize the license plate.*
multiple-image	*There are multiple input images of each car that can be used to recognize the license plate.*
interlaced	*Intrada ALPR supports interlaced input images*
non-interlaced	*Intrada ALPR supports non-interlaced input images*
reference frame	*The customer uses a reference frame to inform the Intrada ALPR product of the image distortion caused by the camera angle.*
8bit images	*The input images contain 8 bits per channel per pixel*
12bit images	*The input images contain 12 bits per channel per pixel*
NL	*Intrada ALPR can recognize Dutch license plates.*
RC	*Intrada ALPR can recognize Taiwanese license plates.*
X_XX	*Intrada ALPR also recognizes whether the license plate is one from an unsupported country.*
LINUX	*The Intrada ALPR binary runs on a Linux system.*
WINDOWS	*The Intrada ALPR binary runs on a MS Windows system.*
WIN_CE	*The Intrada ALPR binary runs on a MS Windows CE system.*
SOLARIS	*The Intrada ALPR binary runs on a Sun Solaris system.*
…	…

Quality attribute name	***Description***
Recognition-rate	*The percentage of license plates that are successfully recognized.*
Error-rate	*The percentage of license plates that are recognized incorrectly.*
Maximum processing time	*The maximum time the ALPR system can spend on processing one input image.*
…	…

Product Scenario Overview	*See Figure 72*

Product Scenarios Description	*Product H5*	*Features:* *single-image, non-interlaced, 12bit, NL, B, GB, CH, F, X_XX, LINUX, …* *Quality:* *Error-rate < 0.1%* *Recognition-rate > 90%* *Maximum processing time < 2 sec.* *…* *Notes:* *This system will handle high quality input images* *…*
	Product L5	*Features:* *single-image, non-interlaced, 8bit, RC, WINDOWS, …* *Quality:* *Error-rate < 1%* *Recognition-rate > 95%* *…* *Notes:* *The system should also recognize license plates during a monsoon. A monsoon causes such large droplets being visible in the image, that they could be recognized as being parts of a character.* *…*
	...	...

12.4.3. Activity 3c - Construct model

Next, the assessment team builds up the Required Variability Model, by using the Provided Variability Model from Activity 2e, the Scoped Feature Dictionary and Product Scenarios from Activity 3b, and the internal variability requirements in the Required Variability Sources from Activity 3a. During this Activity, the assessment team may find that parts of the provided variability model are too incomplete to continue the assessment. In that case, the previous Step is iterated. Otherwise, this Activity continues as follows.

The starting point for the Required Variability Model is the Provided Variability Model from Activity 2e that is subsequently enriched with information from the Product Scenarios from Activity 1b. To accommodate for information about required variability, the COVAMOF Meta-model (see Chapter 8) is extended with Product Scenario entities, Required Change entities and New Object tags (see Figure 73). To build up such a required variability model, the assessment team performs for each product scenario the sub-Activities we describe below. Throughout these sub-Activities we explain the meaning and usage of the new entities and relations in the meta-model (these new entities marked by the dashed lines in Figure 73):

- *Create Product Scenario*: First, for each product scenario, a Product Scenario entity is created in COVAMOF-VS. This Scenario specifies the product group it belongs to, and the time at which the product that is defined by this scenario was, or should be delivered.

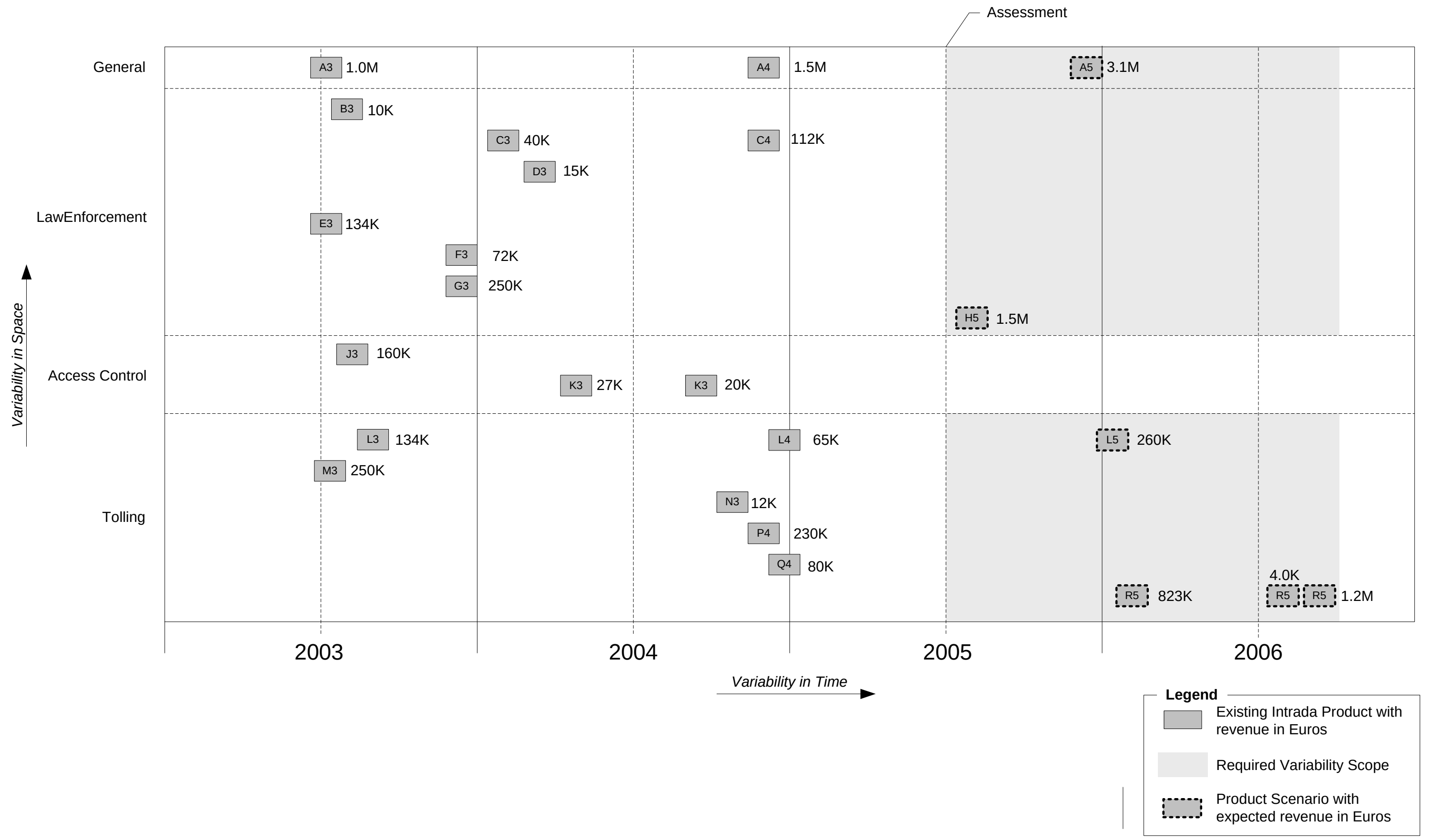


Figure 72. Product Scenarios. This figure shows an overview of the product scenarios that were constructed during Activity 3b, including the expected revenue. Note that while only depict one product scenario in 'General', it actually denotes multiple generic product scenarios that are expected to be sold throughout the Time Scope of the assessment.

- *Identify new variation points and variants:* For the features in the product scenario that are not already a Variant at a Variation Points in the feature layer of the current required variability specification, a new Variant has to be specified. If this new Variant cannot be a Variant of an existing Variation Point, a new Variation Point has to be created as well. These new entities are tagged as "New" in the required variability model (see also Figure 73). Note that new Variants and Variation Points required for multiple product scenarios only have to be created once in the model.
- *Mark variants, and specify values:* Each required feature then translates in a Variant or value that has to be selected for a Product Scenario. To record these selections, Product Scenarios are configured using the COVAMOF-VS configuration assistant. Both the new and Variation Points and Variants from the provided variability model can be configured.
- *Specify variation point attributes:* Based on the variability within the product scenarios, the earliest and latest required binding time, and required type per Variation Point have to be specified. COVAMOF-VS allows for specifying these attributes in the new Required Change entities. These entities contain a reference to the associated COVAMOF entity together with the property-name and the required value.
- *Specify required quality:* In COVAMOF, the important quality attributes are specified using Dependency entities (see also Chapter 8). To specify any requirements on these quality attributes, Required Quality entities have to be created. These entities specify, for one Dependency, the required value, range, maximum or minimum quality for the associated Product Scenario. If the quality attribute was not already part of the model, a new Dependency has to be created first.

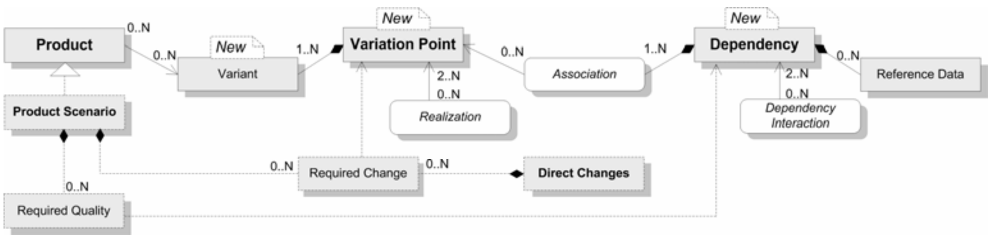


Figure 73. Meta-model of the COVAMOF Required Variability Model. The entities related to required variability are marked by the dashed lines.

In addition to the individual product scenarios, also the internal variability requirements are integrated into the required variability model. These internal variability requirements originate from Activity 3a and are used to specify variability that is not necessarily of interest to the customers, but is required to improve the business processes such as product derivation and maintenance. These requirements often cross-cut the set of product scenarios, and are specified *directly* in the required variability model as Required Change entities that are grouped in a

Direct Changes entity (see also Figure 73). These entities specify new and obsolete Variation Points and Variants, type, binding time, mechanism, and closing time.

Note that the result of this Activity is a model of the required variability in terms of the configuration of the provided variability with the addition of a *delta*: the required variability is specified in terms of provided Variants that need to be bound in the individual product family members, plus a difference to the provided variability in terms of new and obsolete Variation Points and Variants, and changes to the attributes of existing Variation Points.

Table 37. Output of Step 3. Activity 3c: Construct model

Output Name	Representation	Usage
Required Variability Model	COVAMOF required variability model which is a provided variability model with additional required entities. For each product scenario, required Variants and values, required property changes, and required quality.	Step 4

Dacolian Case Study: *Figure 74 shows where the Product Scenarios from the previous Activity required changes to samples from the provided variability model presented in Figure 69. Dacolian accommodated the Image Type Variation Point with the new required image formats identified in the previous Activity (i.e. RAW Bayer and JPEG2000). Similarly, they extended the set of languages with the new Taiwan (RC) and Swiss (CH) Variant entities, and extended the Operating System Variation Point with the new Variants for the four additional Linux distributions. Finally, they modeled the choice between using eight or twelve bits per channel by introducing the new runtime Variation Point Bits per Channel together with its two Variants 8 bits and 12 bits. Note that, although they expected a relation between this new Variation Point and the Error Rate, this relation is not specified during this Activity. Instead, the presence of such association is determined during the next Step, where the assessment team develops solutions to realize these new Variation Points.”*

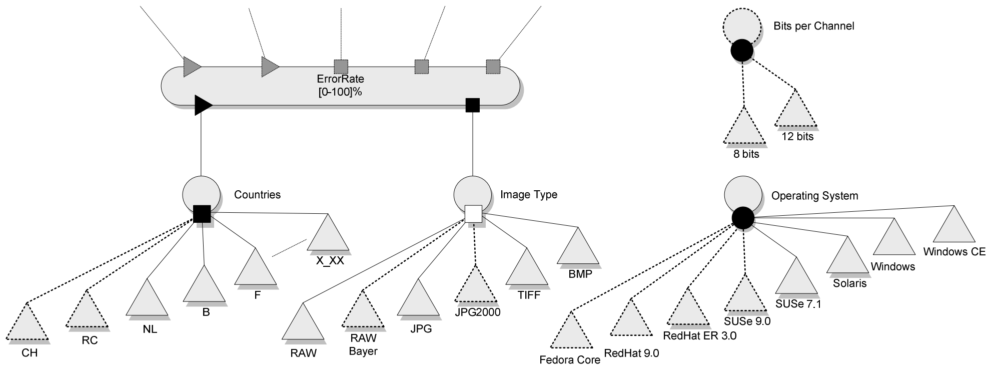


Figure 74. A sample from the Required Variability Model from the Dacolian B.V. case study. COVAMOF-VS not only maintains these new entities, but also keeps track of the required Variants and Dependency values for each Product Scenario. Which Variants are required by which Product Scenario is not shown in this figure.

12.5. Step 4 - Variability Evaluation

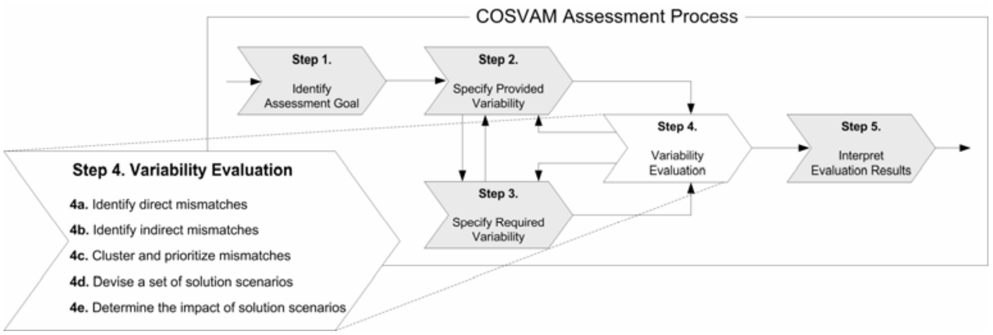

COSVAM Assessment Process
Step 1. Identify Assessment Goal
Step 2. Specify Provided Variability
Step 3. Specify Required Variability
Step 4. Variability Evaluation
Step 5. Interpret Evaluation Results
Step 4. Variability Evaluation
4a. Identify direct mismatches
4b. Identify indirect mismatches
4c. Cluster and prioritize mismatches
4d. Devise a set of solution scenarios
4e. Determine the impact of solution scenarios

Figure 75. Variability Evaluation. In the fourth Step of COSVAM, the required variability model is analyzed in five subsequent Activities to identify mismatches between the provided and required variability. This Activity furthermore specifies the impact of a set of possible modifications that address these mismatches.

The fourth Step in the assessment is the variability evaluation. The purpose of this Step is to determine the impact of changes that are required to solve mismatches between the provided and required variability. The assessment team start this Step by identifying the mismatches (Activity 4a,b), which are clustered into impact analysis sets (Activity 4c). Impact analysis sets consist of mismatches for which the solutions are related. For each of the impact analysis sets, they then devise several solutions (Activity 4d). We refer to these as Solution Scenarios. Finally, the team determines the impact of each of these Solution Scenarios (Activity 4e).

12.5.1. Activity 4a - Identify direct mismatches

The purpose of the first Activity in this Step is to identify the direct mismatches in the Required Variability Model the assessment team created in Activity 3c. These mismatches correspond to the additional entities and required property changes that are explicitly specified in this model. The mismatches do not require propagating choices through the variability model. Direct mismatches that originate from the Product Scenarios are new Variation Points and Variants. Direct mismatches that originate from internal requirements are obsolete Variation Points and Variants, mismatches between binding time, closing time, type and mechanism.

Table 38. Output of Step 4. Activity 4a: Identify direct mismatches

Output Name	**Representation**	**Usage**
Direct Mismatches	Table that identifies and describes the direct mismatches, and for each mismatch the products that are responsible for the mismatch. This can be directly generated from the required variability model.	Activity 4c

Dacolian Case Study: *The external requirements for the binding time of Variation Points in the Intrada ALPR product family are 'do not care', so there were no additional binding time mismatches through the comparison of attributes. The identification of direct mismatches therefore only involved non-existing Variation Points and Variants. For a sample of the output of this Activity, see Table 39.*

Table 39. Samples from the output of Activity 4a in the Dacolian Case Study

Output Name	***Value***		
	Mismatch	***Type***	***Product Scenarios***
Direct Mismatches	*Bits per channel*	*Nonexistent Variation Point*	*Product L5, Product H5*
	8bits	*Nonexistent Variant*	*Product L5*
	12bits	*Nonexistent Variant*	*Product H5*
	CH	*Nonexistent Variant*	*Product H5*
	…	…	…

12.5.2. Activity 4b - Identify indirect mismatches

Where the previous Activity involved the identification of the straight-forward, direct mismatches, the purpose of this Activity is to identify indirect mismatches, i.e. the mismatches that require propagating choices through the variability model. This Activity breaks down in a number of separate actions that are supported by the COVAMOF-VS inference and validation engine:

- The assessment team has to trace Realization Relations between Variation Points to identify how choices at the feature level translate to choices at architecture and component levels. From the realization rules at these Realization Relations, the assessment team (or in this case, tool) can deduce the mismatch where no Variant is implemented for a certain combination of Variants at a higher level. Note that as a result, required Variants are identified at lower levels. These are added to the Required Variability Model. This Activity therefore causes iterations between the Variability evaluation and Specifying required variability Steps (see Figure 75).
- The team furthermore inspects Dependencies to determine whether conflicts occur between Variants, or whether required quality is met after propagating the choices. While conflicts can typically be directly calculated from the COVAMOF model, determining whether required quality is met is more complicated. Depending on how much knowledge is externalized in Step 2, the COVAMOF model contains Reference Data, Associations, and Dependency Interaction entities that can assist engineers in estimating the Dependency value (see also Chapter 8). If there is not enough information in the model yet, this Activity can also cause iterations between the evaluation and specifying provided variability Steps (see also Figure 75).
- Finally, the assessment team inspects whether all product scenarios use the same Variant at Variation Points, to identify *potentially* obsolete Variation Points (in contrast to Variation Points that are required to be obsolete).

Table 40. Output of Step 4, Activity 4b: Identify indirect mismatches

Output Name	**Representation**	**Usage**
Indirect Mismatches	Table that identifies and describes the indirect mismatches, and for each mismatch the products that are responsible for the mismatch. This can be generated by propagating choices through the variability model.	Activity 4c

Dacolian Case Study: *First, the Realization Relations were used to map the feature layer choices and attributes to choices and attributes at the architecture and component layer Variation Points. During the specification of the provided variability model, the assessment team identified that there were almost no exclude or requires Dependencies between Variants in the Intrada ALRP product family, and the realization between combinations of features and choices in the architecture and components should be straightforward. The mapping also did not result in the identification of Variant mismatches on lower levels, but the assessment team did identify two obsolete variation points that were related to input image transformation and tuning the recognition process.*

The identification of indirect mismatches therefore primarily focused on mismatches between provided and required quality for the combinations of country *variants in the product scenarios. For a number of these product scenarios, the analysis was already available. These analyses were performed during the lifecycle of the previous release, when potential customers required a particular product for which the quality was not met. If it would have been met, these products would already have been sold. Now, they ended up in a product scenario.*

Whether the quality could be met for the remaining five product scenarios was determined by testing a configuration that approximated the product scenarios, and assessing the product family architecture. Obviously, this approximation did not incorporate new required features. The test and architecture assessment revealed a number of mismatches. For example, for the scenarios that required multiple country variants, it would not be possible to derive a product from the product family, where the Error-rate was below the required 0.1% with a Recognition rate of 90%. In addition, the Maximum Processing time could also not be achieved for those Product Scenarios. For a sample of the output of this Activity, see Table 41.

Table 41. Samples from the output of Activity 4b in the Dacolian Case Study

Output Name	***Value***		
	Mismatch	***Type***	***Product Scenarios***
Indirect Mismatches	*Error rate of < 0.1% cannot be obtained with 90% recognition rate for the product scenarios that require multiple country variants in one product*	*Required Quality*	*Product Scenario L5, H5, …*
	Processing time can not be met when a product scenario requires more than 1 country variant	*Required Quality*	*Product Scenario L5, H5, …*
	…	…	…

12.5.3. Activity 4c - Cluster and prioritize mismatches

The results of Activity 4a and 4b are lists of mismatches, i.e. the Direct Mismatches and Indirect Mismatches. The goal of Step 4 is to find solutions for these mismatches. On the one hand, de task of developing one big solution for the whole set of mismatches would become too complex for the team. On the other hand, solving all mismatches separately is not possible either, as the solutions for the mismatches may influence or even conflict with each other. We refer to mismatches whose solutions will probably influence each other as *interacting* mismatches.

The goal of this Activity is to find a trade-off between the approaches presented above, by clustering the set of mismatches into separate disjunctive subsets, called impact analysis sets. More precise, the clustering strives for creating impact analysis sets, where mismatches in one impact analysis set can interact with each other, but mismatches in separate sets do not. To determine whether mismatches interact, the assessment team uses the following pointers:

- The first indication that two mismatches may be interacting is whether the mismatches involve Variation Points that both are associated to the same Dependency. Changes to these Variation Points may both influence the Dependency, and are therefore likely candidates to be clustered into one impact analysis set.
- The second indication is directed to the design of the product family artifacts. When two mismatches will be solved in the same part of the product family (e.g. the same source file or database table), there is a higher chance that they will interact, and are therefore candidates to be clustered into one impact analysis set.
- The last indication is the type of mismatch (e.g. a new dependency value or a change of binding time) and the expertise required to solve mismatches. As one expert may be necessary to solve certain mismatches, it can be convenient to cluster these mismatches in one set.

Note that due to, for example, Dependencies that span multiple Variation Points, it may not be possible to construct totally independent impact analysis sets. After the assessment team has finished clustering, they therefore prioritize the impact analysis sets according to the likelihood that the solutions that are proposed for the analysis sets, affect solutions for the other sets.

Also note that clustering and prioritization may need to continue during the evaluation, when new relations are discovered and created between Variation Points. Impact Analysis Sets may furthermore be combined or split if, during the evaluation, it turns out they are respectively more or less related than initially expected.

Table 42. Output of Step 4, Activity 4c: Cluster and prioritize mismatches

Output Name	Representation	Usage
Impact Analysis Sets	A prioritized set of tables of mismatches, and for each table, a description of why the mismatches are grouped and prioritized in that IAS.	Activity 4d

Dacolian Case Study: *The mismatches from the previous two Activities were clustered into twelve Impact Analysis Sets (IAS). Based on the Variation Points that were involved, and the impact of solutions on other solutions, the assessment team first clustered the mismatches related to error-rate, the required image types and bit-levels, plate styles, obsolete variation points, processing time, and country* *variant mismatches in different impact analysis sets. The country variant IAS was further split in four Impact Analysis Sets based on the fact that they involve countries with different plate layouts (such as Arabic, North-American, or Western European), and thus require different developers to address these mismatches. Table 43 shows a sample from the output of this Activity.*

Table 43. Samples from the output of Activity 4c in the Dacolian Case Study

Output Name	*Value*			
	Id	***Priority***	***Mismatches***	***Rationale***
Impact Analysis Sets	*IAS1*	*1*	*Error-Rate too high for multiple country variants*	*The solutions we propose for these mismatches will most likely have an affect on almost all solutions we propose for a few of the other Impact Analysis Sets*
	…	…	…	…
	IAS7	*4*	*Bits per channel, 8bit, 12bit*	*We expect this IAS can be solved independent from the other sets*
	IAS8	*4*	*New Image input RAW Bayer*	*We expect this IAS can be solved independent from the other sets*
	IAS9	*4*	*New Image input JPEG2000*	*We expect this IAS can be solved independent from the other sets*
	...	...	...	...

12.5.4. Activity 4d - Devise a set of solution scenarios

In most cases, several solutions exist for accommodating the variability mismatches in the Impact Analysis Sets. In COSVAM, possible solutions are called Solution Scenarios. Important differences in these Solutions Scenarios are, for example, the decision to solve mismatches product specifically, or to solve them in the product family. Other important differences are the binding time and realization mechanism that are chosen, as well as differences in terms of Dependency values that can be achieved or Dependencies that are removed or created. The purpose of this Step is therefore to identify, for each Impact Analysis Set, the affected entities (product family artifacts, and entities from the variability model) under different Solution Scenarios for the IAS, and to determine the necessary changes on these entities.

As with most design processes, this Activity is highly dependent on expert knowledge and creativity of architects and designers. It can be supported by several techniques, however. Source code analysis, for example, can be used to determine affected entities when replacing stable with new components, or changing variability realization mechanisms. In addition, several examples of realizations of variability have appeared, such as Schnieders and Puhlmann (2006). In addition, Svahnberg et al. (2005), and van Gurp and Savolainen (2006), present a pattern-like catalog of variability mechanisms, with the intent, motivation, solution, constraints, consequences, and examples. Fritsch et al. (2002) on the other hand, present a process with which an organization can create their own catalog of variability mechanisms and their qualities.

Next to the existing body of knowledge, also the variability models that are produced in earlier Steps of COSVAM can be used during this Activity. For Variant mismatches at existing Variation Points or binding time mismatches, for example, the provided variability model can be used to trace Realization Relations to identify the locations where the Variation Point is implemented, as well as which realization mechanisms are used. Dependencies in the provided variability model are furthermore used to identify which parts of the product family in the Abstraction Layer and Subsystem Scope are affected by a change.

Table 44. Output of Step 4, Activity 4d: Devise a set of solution scenarios

Output Name	**Representation**	**Usage**
Solution Scenarios	A set of design documents for each impact analysis set. Each design document in the set is identifiable, and specifically focused on a particular solution.	Activity 4e

Dacolian Case Study: *For almost all Impact Analysis sets created during the previous Activity, the assessment team designed three or four solutions. Most time, about three weeks, was spent on investigating different solutions for the IAS with priority 1, i.e. Error-rate. Only for the Impact Analysis with the lowest priority two solutions were proposed, as the assessment team did not have time to explore other solutions. A sample from the output of this Activity is depicted in Table 45. It describes the solutions proposed for Impact Analysis Set 7, 8, and 9 (see also Table 43). The technical details of these solutions are left out of this thesis.*

12.5.5. Activity 4e - Determine the impact of solution scenarios

The Solution Scenarios specified in Activity 4d imply changes to the product family artifacts. These changes have an impact in terms of the persons or business units that are involved in applying the changes, the time and effort that is required to perform the changes, as well as the effort and time-to-market of the Product Scenarios. The goal of this Activity is to determine these impacts for all Solution Scenarios, so that they can be used to decide which Solution Scenarios should be selected in the next Step.

For the development of the product family, the assessment team determines the product family artifacts that are affected by the changes, and the resources (e.g. people and tooling) that are required to implement the changes. They furthermore estimate the time and effort that will be needed to realize the Solutions Scenario in the product family artifacts.

Table 45. Samples from the output of Activity 4d in the Dacolian Case Study

Output Name	***Value***		
Solution Scenarios	***Id***	***Solutions***	***Description***
	…	…	…
		…	…
	IAS7	*SS7.1*	*The first solution involves changing the entire Image toolbox component so all operations (save, display, analysis, and transformations) in this component can handle both 8bits and 12bits per channel image representations.*
		SS7.2	*The second solution involves only accepting 8bits and 12bits per channel image representations as input, and then translating the input to an 8bits per channel internal image representation. This translation will degrade the recognition performance*
		SS7.3	*The third solution involves only adapting the first steps of the license plate recognition process, so that the detailed 12bits per channel image representation can be used. For later steps this image representation is converted to 8 bits per channel*
		SS7.4	*We will solve this issue product specifically, and only if it is really necessary for an important customer. The ALPR module will then be delivered with a customer specific 8 to 12 bit converter, which customers can use to supply the ALPR module with an 8bits per channel image.*
		Addendum to all solution scenarios	*All image formats that are currently available in our product family support 12bits per channel, except that in the provided variability of the JPEG library, this has to be bound at compile time. The systems need to be able to select between the images types at runtime, however (see also the required variability model). This requires us to add two compiled versions of the JPEG library to our Image toolbox, one for each type.*
	IAS8	*SS8.1*	*We could include the JPEG 2000 library in our Image toolbox component. This will result in an additional DLL, as the JPEG 2000 library cannot be linked with our own code.*
		SS8.2	*We can also decide not to support the JPEG2000 format.*
	IAS 9	*SS9.1*	*The RAW Bayer image format is a complicated format. Our investigation of standard RAW Bayer COTS converters revealed they cannot be used as they create image artifacts that interfere with the image recognition. We could create our own converter.*
		SS9.2	*We can also decide not to support the RAW Bayer format.*
	…	…	…

In addition to the product family artifacts themselves, the team specifies the impact on the product described by the Product Scenarios. To this purpose, the assessment team determines the time and effort that is required to derive the product, given a product family where a particular Solution Scenario has been applied. Where Solutions Scenarios that completely cover the functionality required in product only imply configuration effort, Solutions Scenarios that do not support the required functionality imply product specific adaptations or exclude the product completely.

Note that while COSVAM only prescribes affected entities, time, and effort as impact, the list can be extended with organization specific impacts, such as the impact on tooling for validation and testing.

Table 46. Output of Step 4, Activity 4e: Determine the impact of solution scenarios

Output Name	Representation	Usage
Solution Scenario Impacts	A table that specifies the impacts of each Solution Scenario. These impacts at least encompass the affected artifacts and the time, resources, and effort required for the change as well as the extra effort required for each product scenario during product derivation.	Activity 5b

Dacolian Case Study: *For each of the Solution Scenarios, the assessment team created a table of impacts such as depicted in Table 47. This Table shows the impact for the Solution Scenarios of IAS7, 8, and 9 (see also Table 43 and Table 45). This Impact Analysis Set contained mismatches related to the image formats and bits per channel variants. In addition to effort, resources and affected artifacts, the assessment team determined the impact of Solution Scenarios on Solution Scenarios of other Impact Analysis Sets. For the Error-rate Impact Analysis Set (IAS1), for example, they determined that, depending on the solution, the Solution Scenarios of the four Country Impact Analysis Sets (IAS3-6) were affected by a 1 Person Week effort.*

Table 47. Sample from the output of Activity 4e in the Dacolian Case Study. This sample involves variability that is bound by the system at run-time, so no product derivation effort is listed for these Solution Scenarios.

Output Name	***Value***		
Solution Scenario Impacts	***Id***	***Impact***	***Value***
	…	*…*	*…*
	SS7.1	*Product Family Effort and Resources*	*1 Person Month for Developer 1* *2 Person Months for Developer 2 and 3*
		Affected Artifacts	*The entire Image toolbox component, and all components that use this toolbox (more than 80% of the design)*
	SS7.2	*Product Family Effort and Resources*	*1 Person Week for Developer 1*
		Affected Artifacts	*The Image Toolbox component.*
	SS7.3	*Product Family Effort and Resources*	*2.5 Person Weeks by Developer 1* *2 Person Weeks by Developer 3*
		Affected Artifacts	*Image Toolbox and Segmenter component*
	SS7.4	*Product Family Effort and Resources*	*None (handled product specifically)*
		Affected Artifacts	*None (handled product specifically)*
	SS8.1	*Product Family Effort and Resources*	*1 Person Day for Developer 1*
		Affected Artifacts	*Installer, and Read operation of Image Toolbox component*
		Licensing	*EUR 50 for each configuration with a JPEG 2000 library*
	…	*…*	*…*

12.6. Step 5 - Interpret the evaluation results

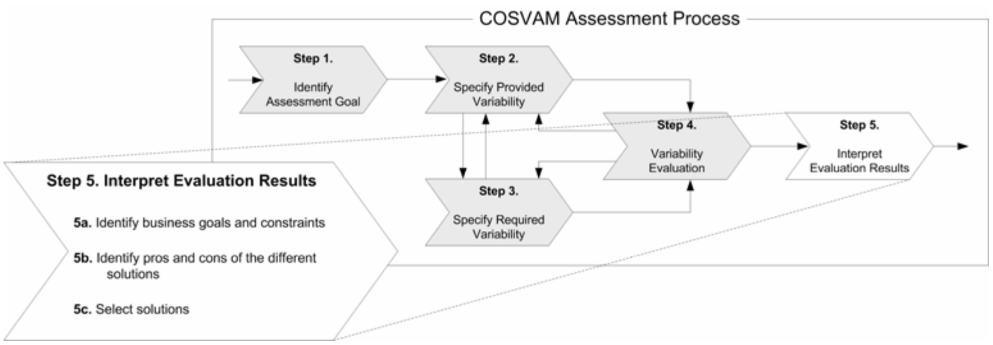

COSVAM Assessment Process
Step 1. Identify Assessment Goal
Step 2. Specify Provided Variability
Step 3. Specify Required Variability
Step 4. Variability Evaluation
Step 5. Interpret Evaluation Results
Step 5. Interpret Evaluation Results
5a. Identify business goals and constraints
5b. Identify pros and cons of the different solutions
5c. Select solutions

Figure 76. Interpret Evaluation Results. The fifth Step of the COSVAM assessment process involves analyzing and rating the solutions from Step 4 to select the most appropriate ones as the assessment outcome.

The purpose of the interpretation is to draw conclusions based on the evaluation. As we explained in the introduction of this section, we limit the discussion in this chapter to the situation in which the optimal solution scenarios need to be selected for release planning (see Activity 1a). As we show in Figure 76, in this final Step of COSVAM, assessment team *identifies relevant business drivers and constraints (Activity 5a), identifies pros and cons of different solutions (Activity 5b)*, and *selects solutions (Activity 5c).*

12.6.1. Activity 5a - Identify business goals and constraints

The interpretation starts with identifying aspects that have an influence on which solution should be chosen, e.g. with respect to the Expected Results specified in Activity 1b. For identifying the optimal solution scenarios these are aspects that constrain possible solutions, or that help in achieving certain goals. The goals and constraints can have a technical, economical, or organizational background, but usually involve a combination of all three. Examples of constraints are the size of development teams, and the maturity of technology. Examples of goals are reduction of testing effort, and improved performance, ease of maintenance, time-to-market, and market share. See, for example, Bosch (2000), and Clements and Northrop (2001).

Table 48. Output of Step 5, Activity 5a: Identify business goals and constraints

Output Name	Representation	Usage
Goals and constraints	A table that lists the name, description, and (where possible) the value of the goals and constraints.	Activity 5b

Dacolian Case Study: *This Activity identified eight relevant goals and constraints; they are depicted in Table 49. The order in this Table suggests a weighing of the goals and constraints, viz. the Highest Performance, Development Constraints, and Large Projects are the most important goals and constraints, while Open Variation Points and Own Code are less important.*

Table 49. Output of Activity 5a in Dacolian Case Study

Output Name	***Value***	
Goals and constraints	***Name***	***Description***
	A. Highest performance	*We want to be market leader when it comes to performance in terms of recognition and error-rates of our products*
	B. Development Constraints	*Our resources for the changes to Intrada ALPR are five Months, and four Developers.*
	C. Large Projects	*Large Tolling or Law Enforcement projects create a lot of revenue, so their requests have priority of customers that only buy a small amount of products.*
	D. Long term customer relationships	*Customers that bought products in the previous years continue to buy new products. Requests from these customers have priority over new customers.*
	E. Configurable Product Family	*To be able maintain this product family with a minimal amount of developers, we strive for one binary that can be configured for each customer., where no product specific changes are necessary.*
	F. Binding time at runtime	*If the system can configure itself at runtime, no product derivation effort is required. We prefer this over variability that has to be handled pre-deployment.*
	G. Open Variation Points	*Based on our experience, we strive for variation points that are open to extension with new variants, so that our product family is more future proof.*
	H. Own Code	*We prefer own code over COTS, as it has proven to be hard to have control over the quality of COTS components (the past has proven they cause unexpected pop-ups and errors, and are not as platform independent as often claimed).*

12.6.2. Activity 5b - Identify pros and cons of the different solutions

A set of Solutions Scenarios that contains exactly one Solution Scenario for each of the Impact Analysis Sets forms a possible outcome of the assessment. In COSVAM, these sets are referred to as Solution Sets. The purpose of this Step is to find the Solution Set that optimizes the business goals and constraints identified in Activity 5a. This and the next Activity address the following two problems. First, judging the optimum for a set of goals and constraints is a multi-objective decision problem, i.e. solutions will generally be good for one set of goals and constraints, but bad for the others. Second, determining the optimum for all possible Solution Sets suffers from combinatorial explosion, i.e. since each Impact Analysis Sets has multiple Solution Scenarios, the amount of possible Solution Sets increases exponentially.

COSVAM deals with these problems in the following way. In this Activity, each Solution Scenario is rated independently on each business goal and constraint identified in Activity 5a, instead of as combinations of Solution Scenarios. This rating is, amongst others, based on the Solution Scenario Impacts determined in

Activity 4e. In the next Activity, the overview of pros and cons of each Solution Scenario is used to find the optimal Solution Set (see Activity 5c). The rationale for first rating the Solution Scenarios independently, is that from the Solution Scenarios for each Impact Analysis Sets, typically only a small amount of Solution Scenarios will sufficiently address the goals and constraints to be likely candidates for a Solution Set.

Table 50. Output of Step 5, Activity 5b: Identify pros and cons

Output Name	Representation	Usage
Overview of Pros and Cons	For all Solution Scenarios of an Impact Analysis Set, a Table that analyses the pros and cons of the scenarios. To this purpose each Solution Scenario is rated on all goals and constraints identified in Activity 5a.	Activity 5c

Dacolian Case Study: *In the case study at Dacolian, the assessment team created an overview of pros and cons for the Solution Scenarios of all Impact Analysis Sets. To this purpose, the assessment team either used an exact value or a scale ranging from – to 0, to ++. The – denotes a situation where the Solution Scenario negatively influences the goal or conflicts the constraint, while ++ denotes a situation where the Solution Scenario contributes to the goal, and does not conflict the constraint.*

12.6.3. Activity 5c - Select solutions

In this final Activity, the Overview of Pros and Cons identified in previous Activity are weighed, and the solutions that optimize the constraints and business goals are selected. To this purpose, the assessment team first constructs candidate Solution Sets by selecting the Solution Scenario candidates for each Impact Analysis Set. This selection is based on whether the Solution Scenarios have an acceptable score on the pros and cons. These Solution Scenario candidates represent sub-optimal solutions, however. Different combinations of Solution Scenarios from different Impact Analysis Sets represent different overall pros and cons of Solution Sets. In addition, the combination of Solution Scenarios may introduce scheduling issues, and have a different impact on which product scenarios can be facilitated (which translates in a different amount of revenue).

Depending on the number of Solution Scenario candidates per Impact Analysis Set, the assessment team can either determine the impact, scheduling, and pros and cons of all Solution Sets at once, or first select the Solution Scenarios that have the best score for their Impact Analysis Set, and then iteratively determine the optimal Solution Set by exchanging Solution Scenarios in the Solution Set.

Table 51. Output of Activity 5b in the Dacolian Case Study. This table specifies the pros and cons for the Solution Scenarios of Impact Analysis Set 7, 8, and 9. To score the Development Constraint (B), the assessment team depicted the total effort required for changes to the product family (in Person Months).

Output Name	***Value***
Overview of Pros and Cons	*IAS7:*

Solution Scenario/ Goal or constraint	***A***	***B***	***C***	***D***	***E***	***F***	***G***	***H***
SS7.1	++	*3*	+	+	+	+	--	*0*
SS7.2	--	*0.25*	+	+	+	+	++	*0*
SS7.3	+	*1*	+	+	+	+	*0*	*0*
SS7.4	-	*0*	--	--	--	--	*0*	*0*

IAS8:

Solution Scenario/ Goal or constraint	***A***	***B***	***C***	***D***	***E***	***F***	***G***	***H***
SS8.1	+	*0.03*	*0*	+	+	+	*0*	--
SS8.2	-	*0*	*0*	-	-	*0*	*0*	*0*

IAS9:

Solution Scenario/ Goal or constraint	***A***	***B***	***C***	***D***	***E***	***F***	***G***	***H***
SS9.1	++	*1*	+	+	+	+	*0*	+
SS9.2	-	*0*	-	-	-	*0*	*0*	*0*

The result of this Activity is a Solution Set that optimizes the business goals and constraints (see Activity 5a), a description of its impact on the Product Scenarios, and the Motivation of the selection. The assessment is now almost finished. The only remaining task is communicating the output according to the Communication output of Activity 1b.

Table 52. Output of Step 5, Activity 5c: Select solutions

Output Name	Representation	Usage
Selection of Solutions	A list of Solution Scenarios	Assessment Output
Impact of Selection	A table that describes for each product scenario the total required effort (the type as well as the estimated amount). Note that this output is a view on the Solution Scenario Impacts from Activity 4e.	Assessment Output
Motivation	A document motivates the Selection of Solution with the analysis from this Activity.	Assessment Output

Dacolian Case Study: *A sample of the output of the selection process at Dacolian is depicted in Table 53. This table describes the motivation for the selection of a number of Solution Scenarios. From this selection, and the descriptions of the Solution Scenarios, Dacolian constructed the documents to communicate the results internally and externally. We discussed the communication of these results in Activity 1b.*

Table 53. Sample from the output of Activity 5c in Dacolian Case Study.

Selected Scenario	***Motivation***
…	…
IAS7: SS7.3	*As the only Solution Scenario candidate for IAS1 (Error-rate) already consumes 4 MM for both Developer 2, and 3, Solution Scenario SS7.1 is not a good candidate as it causes a scheduling problem (see column B in Table 51) . SS7.2 and SS7.4 are not good candidates either, as the impact on the High performance (column A) is unacceptable. Based on the pros and cons, the only acceptable Solution Scenarios of IAS 7 is therefore SS7.3.*
IAS8: SS8.2	*Although being able to support JPEG2000 through SS8.1 increases performance (Column A in Table 51), the licensing cost of the JPEG2000 is detrimental to our revenue.*
IAS9: SS9.2	*Despite the fact that SS9.1 (RAW Bayer) is a far better scenario than SS9.2, it causes a scheduling problem (see also column B in Table 51). Developer 2 already has to spend all his effort on the other Solution Scenarios in the Solution Set. These other Solution Scenarios are far more important when it comes to revenue.*
…	…

12.7. Evaluation

A strong scientific validation of COSVAM is hard, as this case study does not allow us to verify whether the way Dacolian used to work would actually have produced better results. Eight months after they performed the variability assessment, however, we asked the assessment team from Dacolian for their experiences with COSVAM. Below, we first present the strengths and weaknesses they identified, and then discuss how these experiences relate to the variability assessment issues of Section 11.2.

12.7.1. Experiences

"The assessment took us five weeks, of which three weeks were spent on the analysis of solution scenarios that we did not select for the assessment outcome. In the end, however, we do feel that assessing our product family according to the COSVAM method was a good investment. The experiences we have on working with the new release show that we indeed predicted the required variability very well in Step 3, and that we chose the right solution scenarios to address them. The different alternatives we investigated are certainly not a waste of effort, as we feel they helped us in choosing a set of solutions that really addressed our business goals, while at the same time making sure that we could implement them within the organizational, economical, and technical constraints. When we evaluated our experiences with COSVAM, we identified five main strengths and one weakness".

Strengths

- *Structure:* "In the original situation, we had no structured assessment process to evolve our Intrada ALPR product family to a new version. Instead, the assessment and the actual implementation of necessary changes were not clearly separated. The ad-hoc nature of our assessments often caused situations where we would implement changes and then simply run out of time. COSVAM really helped us in explicitly planning the assessment tasks, by showing how different inputs and outputs could be used to be able select an optimal set of solutions. As the outcome of COSVAM is furthermore structured in separate solutions for the impact analysis sets, we can easily plan the implementation of these smaller parts".
- *Decisions and assumptions made explicit:* "Another benefit of COSVAM is that after the assessment, there is still a clear list of required variability that will not yet be implemented in the product family (such as some required country variants). For those aspects, we did have the solution scenarios prepared, tough. In the end, this allowed us to quickly implement some scenarios that were rejected based on scheduling issues, but that could be implemented when it turned out some developers required a bit less effort than estimated".
- *Optimal solutions for actual problems:* "We used to identify required variability in an ad-hoc manner. In addition, the decision to implement a particular change was mainly based on technical challenges we would like to tackle, because, honestly, those were the changes the developers thought to be fun. With the instructions provided by COSVAM, we were forced to, and

assisted in making decisions that represent value to our business, rather than our personal preferences."

- *Maturity:* "Based on the instructions we received on the method, we were confident enough to apply the method by ourselves. After completing the assessment, we still think COSVAM is a mature and complete technique that makes sure no necessary Steps are forgotten, and that clearly specifies the required inputs and outputs".
- *COVAMOF:* "The final main strength of COSVAM is its approach to modeling variability. In the past, experiences with having no variability model frequently resulted in situations where mismatches were not found. Having such a model in place makes it a lot easier to identify variability mismatches, as it helps to keep an overview of all choices and dependencies. Another benefit of using the COVAMOF externalization process is that taking the information from different sources revealed that different experts actually had different insights into the dependencies between choices, and that some experts had assumptions about the provided variability that turned out to be wrong. Creating such a model thus improved our common understanding of the provided variability of the Intrada ALPR product family".

Weaknesses

- *Focused on larger organizations:* "When we look at COSVAM as a whole, we think that Step 1 is now a bit superfluous for small organizations like Dacolian. While we do see the benefit in case an external assessment team is involved, in particular in situations where internal people perform the assessment, the identification of the assessment team, influencing factors, etc. feels a bit like overkill. We would assume these aspects to be known to all people involved in the assessment".

"The evaluation thus concluded that experiences of applying COSVAM, as well as the experiences with its results, strengthen our decision to continue to apply this method for planning our product family releases."

12.7.2. Discussion

The examples and experiences presented in this chapter are related to the variability assessment issues we discussed in the previous chapter. Below, we discuss how the examples and experiences cover these issues.

Methodological issues

- *Unstructured:* The assessment team concluded that COSVAM divides the assessment problem into manageable pieces, where the processes and their results can be carefully planned (see Strength: Structure).

- *Reactive instead of proactive:* The experiences of Dacolian showed that by taking future products into account, the assessment team was able to create a product family release that turned out to match the requirements of the products that had to be built on top of that release (see, for example, Strength: Optimal solutions for actual problems).
- *Generalized decisions:* A few months after the assessment, the experiences at Dacolian showed that now, Solution Scenarios were selected that actually represent value to their business (see, for example, Strength: Optimal solutions for actual problems).
- *Lack of removing obsolete variability:* The assessment team was able to identify two obsolete variation points at the feature level, and to make an educated decision whether to remove them (see, for example, Activity 4b).

Knowledge issues

- Addressing only one layer of abstraction: The assessment at Dacolian encompassed an assessment on all layers of abstraction (see Activity 1c). This allowed the assessment team to identify detailed changes that were required to the variability provided by the product family artifacts.
- Implicit variability: The experiences at Dacolian show that even the incomplete model that was constructed during the limited time of the assessment, created a better understanding of the dependencies between variation points (see Strength: COVAMOF).
- Neglecting implementation dependencies between features: The implementation dependencies that were captured by the provided variability model allowed the assessment team to cluster the mismatches, so that they could explicitly relate the impact of Solution Scenarios to each other (see, for example, the impact of the Error-rate Impact Analysis Set in Activity 4e).
- Insufficiently exploring alternative solutions: Finally, the assessment concluded that explicitly considering multiple Solution Scenarios helped them in choosing a set of solutions that really addressed their business goals (see Section 12.7.1).

12.8. Conclusion

The predominant challenge, in most software product families, is the management of the variability required to facilitate the product differences. Over time, the variability required from the product family evolves, but assessing how the variability provided by the product family artifacts needs to evolve in response, is a non-trivial problem. Currently, however, there is a lack of techniques that specifically support the assessment of software variability in product family artifacts. In response to this, we have developed COSVAM, the COVAMOF Software Variability Assessment Method.

12.8.1. Contribution

The main contribution of COSVAM is that it is the first technique for assessing variability with respect to the needs of a set of product scenarios. The 5 Steps of COSVAM form a technique that can be tuned to optimally address a variety of situations where the question of whether, how and when to evolve variability is applicable. It defines different results that can be produced, such as an overview of mismatches, different solution scenarios and a selection of the optimal solutions, as well as defines the way in which information has to be selected and interpreted to produce those results. It furthermore provides a means to externalize the provided variability of the product family.

The Activities of the five Steps of COSVAM address the issues we presented in the previous chapter as follows:

- *Unstructured:* COSVAM addresses this issue by distinguishing multiple goals, defining repeatable Steps and Activities, and forcing assumptions and decisions to be made explicit.
- *Reactive instead of proactive:* One of the main ideas behind COSVAM is that, for optimally evolving the variability of the product family artifacts, future product specifications are incorporated in the product scenario set (see Activity 3a, and 3b). COSVAM is therefore not only suitable as reactive assessment for benchmarking, identifying mismatches, and determining product specific adaptations, but also as proactive assessment for release planning.
- *Generalized decisions:* Decisions with respect to mismatches between the required and provided variability are grouped in impact analysis sets, i.e. sets of mismatches that are related as they involve the same component, for example. For each impact analysis set, multiple solution strategies are devised that each have a different impact (see Step 4). The interpretation Step (Step 5) then identifies the pros and cons of those Solution Scenarios, with respect to the goals and constraints of the product family. This allows the selection of set of Solution Scenarios that optimally addresses the requirements posed on the product family.
- *Lack of removing obsolete variability:* The COSVAM is a need-based assessment method that compares the variation points, variants, and quality attributes that are required within a set of product scenarios with the variability provided by the product family. Obsolete variability is identified indirectly when none of the product scenarios requires particular variation points, variants or quality, or when they are directly identified by the experts (see Activity 3c, 4a, and 4b). Obsolete variability is marked as potentially obsolete, and during the evaluation and interpretation Steps it is decided whether the obsolete variability will be removed, or whether the removal will be postponed.
- *Addressing only one layer of abstraction:* The COSVAM is built around a notion of variability that treats variation points and dependencies uniformly as

first-class citizens that are related across abstraction layers (see Step 2). The COSVAM is aimed at all abstraction layers, but when required, allows for focusing on a subset of layers and artifacts by selecting the appropriate assessment scope. To minimize the inaccuracy and risks associated with a limited scope, the assessment allows for iterations between different Steps (Step 2, 3, and 4), and weighs risks in the interpretation Step (Step 5).

- *Implicit variability:* COSVAM provides an iterative process that is focused on minimizing effort with respect to externalizing variability that is irrelevant for the assessment (see Step 2). COSVAM explicitly deals with missing information by forcing the documentation of which parts are deliberately left implicit and weighing missing information in the interpretation Step (Step 5). COSVAM furthermore provides a Step-wise process for externalizing provided variability to a variability specification that models variability uniformly
- *Neglecting implementation dependencies between features:* During the evaluation (Step 4), it is verified whether the dependencies in the provided variability allow for offering the required combinations, how different solutions address these implementation dependencies, and how solutions have an impact on each other.
- *Insufficiently exploring alternative solutions:* The COSVAM explicitly deals with this issue by assisting engineers in devising multiple Solution Scenarios, and weighing the pros and cons of these Solution Scenarios (see also above: generalized decisions).

Since it addresses all variability assessment issues discussed in previous chapter, COSVAM represents a structured technique to answer the question of whether, how, and when variability should evolve. COSVAM is focused on an *evolutionary* approach to variability management. The COSVAM is specifically designed as a suitable assessment technique, as long as the number of mismatches does not cross a threshold at which, due to design erosion (Gurp and Bosch, 2002), it is cheaper to completely redesign the product family architecture and components (a revolution). The purpose of institutionalizing and applying the assessment is to delay this point as long as possible.

12.8.2. Acknowledgements

We would like to thank the participants from Dacolian for their valuable input.

12.9. References

Bosch, J., 2000, Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach, Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7.

Clements, P., Kazman, R., Klein, M., 2001, Evaluating Software Architectures, Methods and Case Studies, Addison-Wesley, ISBN 0-201-70482-X.

Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns, SEI Series in Software Engineering, Addison-Wesley, ISBN: 0-201-70332-7.

Fritsch, C.; Lehn, A.; Strohm, T., 2002. Evaluating variabil-ity implementation mechanisms, Proceedings of International Workshop on Product Line Engineering Seattle, pp. 59-64.

Gamma, E., Richard, H., Johnson, R. and Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley. ISBN 0-201-63361-2.

Gurp, J. van, Bosch J., 2002. “Design Erosion: Problems & Causes”, Journal of Systems & Software, 61(2), pp. 105-119.

Gurp, J. van, Savolainen, J., 2006, Service Grid Variability Realization, proceedings of 10th International Software Product Line Conference (SPLC 2006).

Kang K., Cohen S., Hess J., Nowak W., and Peterson S., 1990, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, SEI.

Maccari, A., 2002. "Experiences in assessing product family software architecture for evolution", Proceedings of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, pp. 585-592.

Schnieders, A., Puhlmann, F., 2006. Variability Mechanisms in E-Business Process Families, Proceedings of 9th International Conference on Business Information Systems (BIS 2006), volume P-85 of Lecture Notes in Informatics, Austria, pp. 583-601.

Svahnberg, M., Gurp, J. van, Bosch, J., 2005. A Taxonomy of Variable Realization Techniques, Software Practice & Experience, vol 35, pp. 1-50.

Taguchi, G. and Phadke, M.S., 1984. Quality Engineering Through Design Optimization. IEEE Global Telecommunications Conference, GLOBECOM ´84, Atlanta, GA, pp. 1106-1113.

Conclusion

In this Part, it is time to reflect on the findings we presented in the previous Parts. We therefore summarize these findings, relate them to our research questions, and discuss their validity. Finally, we discuss the ideas for future work.

Chapter 13 Summary and Discussion

In the Introduction to this thesis, we formulated the overall research question as "Which variability management techniques can effectively decrease the application engineering effort in software product families". We divided this question into five main research questions, and spent ten chapters to answer them. In this chapter, we provide the summary of the answers that are provided in those chapters. We furthermore discuss the validity of these answers, and the open issues and improvements that should be addressed in future work.

13.1. Answers to research questions

Together, the answers to the five main research questions provide the answer to our overall research question. In the table below, we show the link between the five research questions, the results we produced by answering these questions, and the chapters in this thesis (see also introduction to this thesis). In the sections below, summarize our answers.

Table 54. Link between research questions, results, and chapters (to assist the reader, this is a copy from Table 3).

Research Question	Result	Chapter
Q1 Overview	Product Derivation Framework	Chapter 3 and 4
Q2 Investigation	Problems and issues	Chapter 5
Q3 Variability Modeling Classification	Classification of variability modeling techniques	Chapter 6
Q4 Variability Modeling Technique	Conceptual view on variability	Chapter 7
	COVAMOF Language	Chapter 8
	COVAMOF Tool-suite	Chapter 8
	COVAMOF Derivation Process	Chapter 9 and 10
Q5 Variability Assessment Technique	COVAMOF Software Variability Assessment Method (COSVAM)	Chapter 11 and 12

13.1.1. Q1 Overview

We formulated the first research question as follows:

Q1 Overview: What terminology, tools and processes are used for product derivation in research and industry?

Answer to Q1: Products are derived in product families that can be classified in two dimensions, i.e. scope of reuse, and domain scope. The scope of reuse denotes to what extent the commonalities between products are exploited. It ranges from standardized infrastructure to configurable product family. The domain scope denotes the extent of the domain or domains in which the product family is applied. It ranges from single product family to product population.

We realized that the processes that are used in the domain of single product families, can be generalized into a generic product derivation process. This generic process consists of two phases, i.e. the initial and the iteration phase. In the initial phase, a first configuration is created from the product family artifacts. During this phase, the application engineer has substantial freedom in choosing alternative product family artifacts. In the iteration phase, the initial configuration is modified in a number of subsequent iterations until the product sufficiently implements the imposed requirements. The freedom of choice of the application engineer is much more limited during the iteration phase as all decisions have to be made within the context of the product configuration at hand. For more details, see Chapter 3. For examples, see Chapter 4.

13.1.2. Q2 Investigation

We formulated the second research question as follows:

Q2 Investigation: Which problems and issues do organizations face during product derivation?

Answer to Q2: The problems and issues that are common among organizations that derive products from a product family are: *False positives of compatibility check during component selection, Large number of human errors, Consequences of variant selection unclear, Repetition of development, Different provided and required interfaces complicate component selection.*

All these problems have two core issues in common, i.e. complexity in terms of the sheer number of choices and dependencies, and properties where relevant knowledge is missing. These issues are not problematic per se, but become problematic by the failure to effectively deal with them. Therefore, we also investigated the causes of complexity and implicit properties, and presented strategies to deal with them. For a detailed discussion of these issues, causes and strategies, see Chapter 5.

13.1.3. Q3 Variability Modeling Classification

To determine how existing variability modeling addressed the issues we identified, the third question involved the classification of existing variability modeling techniques:

Q3 Variability Modeling Classification: How do the variability modeling techniques that have been proposed over the past years support product derivation?

Answer to Q3: We compared and classified five techniques that aim to address these issues, specifically the variability modeling techniques CBFM, VSL, ConIPF, Pure::Variants, and Koalish. We concluded that only one technique had a defined process, and that, while this situation can often not be achieved in practice, most techniques only focus on configurable product families where all knowledge can be fully formalized.

We reached this conclusion by formulating a classification framework that consists of two categories, i.e. modeling and tools. The modeling category focused on how the techniques deal with choices, product models, abstraction, formal constraints, quality attributes, and incompleteness and imprecision. The tools category focused on looking at whether the tools supported views, active specification, configuration guidance, inference, and effectuation. For more details, see Chapter 6.

13.1.4. Q4: Variability Modeling Technique

Based on the classification of variability modeling techniques, we identified the need for new and improved modeling facilities, primarily for the support of the complexity and different types of knowledge. We therefore formulated the fourth question as follows:

Q4 Variability Modeling Technique: What are key ingredients for a variability modeling technique that does address the product derivation issues of complexity and implicit properties?

- ***Which concepts are involved in handling complex dependencies and different types of knowledge?***

  Answer: These concepts breakdown into three categories. The first is related to tacit, documented, and formalized knowledge. The second to imprecise and incomplete knowledge, and the third to dependency interaction. Tacit and documented knowledge, imprecision and incompleteness, and dependency interactions all cause iterations during product derivation. For a detailed explanation and examples, see Chapter 7.

- ***How should these concepts be addressed in a variability modeling language?***

  Answer: These should be addressed by the modeling entities variation points, variants, realization relations, dependencies, associations, reference data, and products. To reduce complexity (see Chapter 5), the variation points are organized hierarchically with realization relations. In addition, dependencies are grouped on the level of variation points, rather than relations between the variants. This reduces complexity by providing a more abstract view on dependencies. It also allows to model dependency interaction as a way to prevent unnecessary iterations (see Chapter 7). Associations and reference data are used to capture less formal knowledge (see Chapter 7), and offer a way to capture knowledge reactively (Chapter 8 and Chapter 10). All these entities are found in our variability management framework COVAMOF. For more details, see Chapter 8.

- ***How should such a modeling language be used during product derivation?***

  Answer: The construction and use of such a model should be supported by a tool, and a process. During our research we constructed a proof of concept tool-suite for Microsoft Visual Studio.NET. The COVAMOF tool-suite extracts modeling information from the files in the Solution, supports different views such as the derivation and dependency view, and is responsible for maintaining consistency between the model and the artifacts in the Solution. Due to the concepts involved in complex dependencies, the derivation process is still iterative and an instance of our generic product derivation process (see also Part I). It consists of five steps, i.e. Product Definition, Product Configuration, Product Realization, and Product Testing. In Chapter 9, we showed how each of these steps translates into the creation and use of entities in the COVAMOF model in the COVAMOF tool-suite.

Answer to Q4: That these ingredients do address the product derivation issues of complexity and implicit properties was shown by our validation of the benefits that COVAMOF offers for product derivation. This validation was based on the Dacolian case we described in Chapter 4. Our hypotheses were that the effort required to derive products would decrease when COVAMOF was used, and that persons other than experts could then also derive products from a product family.

Our validation clearly shows that with COVAMOF, engineers that were not involved in the product family, i.e. non-experts, were now capable of deriving the products in 100% of the cases, compared to 29% of the cases without COVAMOF. In addition, with COVAMOF, the number of iterations required to derive products was reduced by 42% for experts, and 54% for non-experts. The experiment results thus firmly support our hypotheses that (1) experts would be able to derive products faster, and that (2) non-experts would now also be able to derive products (faster). For more details on these experiments, see Chapter 10.

13.1.5. Q5: Variability Assessment Technique

The investigation of the product derivation issues also showed that the complexity of variability is caused by a mismatch between the variability that is provided by product families, and the variability that is required by the products. Our last research question was therefore related to a technique that can be used during domain engineering, i.e. variability assessment.

Q5 Variability Assessment Technique: Is it possible to assess variability so that the mismatch between the provided variability of the product family and required variability of the products can be reduced?

- ***What is variability assessment and what are the issues?***

  Answer: Variability assessment answers the question of whether, when, and how variability should evolve. The issues that are at play are related to a practice where variability assessment is *Unstructured*, *Reactive instead of proactive*, based on *Generalized decisions*, with a *Lack of removing obsolete variability, Addressing only one layer of abstraction,* using *Implicit variability, Neglecting implementation dependencies between features*, and *Insufficiently exploring alternative solutions.* See also Chapter 11.

- ***What can we do to address those issues?***

  Answer: Employ our COVAMOF Variability Assessment Method (COSVAM), a technique that structures the process that is used to determine whether, how, and when variability should evolve into five main steps, i.e. formulating the goal, specifying provided variability, specifying required variability, evaluation, and interpretation. Each of these steps consists of a handful of activities that produce well-defined deliverables (see Chapter 12). The assessment technique builds upon the experience of existing assessment techniques such as ATAM (Clements et al., 2001) and ALMA (Bengtsson et al., 2004). For example, it uses the same overall structure and ways to prepare the assessment, but then tuned to variability. To handle the complexity of variability, it uses COVAMOF models to capture variability, and clusters variability mismatches into impact analysis sets. As organization typically do not have explicit variability model yet, it also defines way to externalize this information. To prevent insufficient, irrelevant and voluminous documentation (Chapter 5), the assessment defines iterations between the specifying provided variability, specifying required variability and evaluation steps. COSVAM further more uses information from the past, and predictions of the future to identify unused variability. It furthermore uses predictions, and weighing alternative solutions to prevent non-optimal realization of variability (see Chapter 5). For more details on COSVAM, see Chapter 12.

Answer to Q5: To answer whether reducing mismatches is indeed possible, we exemplified and evaluated the steps and activities of this method with the results of applying it in the industrial environment. The strengths are its *Structure*, that *Decisions and assumptions are made explicit*, that it focuses on *Optimal solutions for actual problems*, its *Maturity*, and the use of *COVAMOF*. A perceived weakness is that it seems *Focused on larger organizations*. For more details, see Chapter 12.

13.2. Validity

Now that we have presented the results of our research, it is time to reflect on the validity of our conclusions. There are several threats to the validity of the conclusions we present in this thesis. Below, we discuss the threats for each of our research questions.

13.2.1. Q1 Overview and Q2 Investigation

The validity of our product derivation framework and problems and issues is related to the question whether the results of a case study at two organizations cover all relevant issues and whether they can be generalized. To ensure this validity, the case studies involved interviews with key personnel involved in product derivation, at several business units in two organizations. Both companies are large and mature industrial organizations that mark two ends of a spectrum of product derivation; Robert Bosch produces thousands of medium-sized products per year, while Thales Nederland produces a small number of very large products. To prevent overlooking important issues, the interviews were guided by a questionnaire. There was also enough room for open, unstructured questions and discussions. We furthermore recorded the interviews for further analysis afterwards, and used documentation provided by the companies to complement the interviews.

13.2.2. Q3 Variability Modeling Classification

The validity of our classification primarily depends on the categories that constitute our classification framework. As we discussed in Part I, the classification framework misses one important category, i.e. what processes are defined. We left this category out because almost no technique addressed this aspect.

We divided the remaining categories into characteristics. The selection of each characteristic was the result of insights gained from three sources. First, there is an existing body of knowledge regarding software product families and variability management. Second, we compared the underlying concepts of the variability modeling techniques to identify similarities and differences. Finally, a number of case studies and other articles identify a set of issues that should be addressed for variability management during product derivation (including the issues we identified when we answered Q2). In other words, the characteristics of our classification framework reflect the important facets of variability modeling, and everyday practice (see also Part I).

13.2.3. Q4 Variability Modeling Technique

The validity of our conclusion regarding this question is primarily related with the results of the validation experiment we discussed in Part III. With this experiment, we validated whether the use of COVAMOF (1) enables non-experts to derive products and (2) whether engineers derive products faster. The questions regarding validity are whether our measurements are correct, whether the results are statistically significant and whether the results are generalizeable.

We answered the first question about correctness in Part III: correctness is addressed by taking the competence factor into account, involving multiple participants, and making sure experience gained by participants during one part of the experiment does not influence the other parts.

To measure the statistical significance of the reduction of time and number of iterations, we use the t-test. With the t-test, we calculate the chance that the reductions are just a matter of coincidence rather than being caused by introducing COVAMOF. In general, statistical significance is achieved when this chance is below 5% (a p-value below 0.05). The p-value for the reduction of the total time is 0.0040 (df=24 and t=2.89). As this p-value is below 0.05, our conclusion of COVAMOF resulting in a reduced number of hours is statistically significant. The p-value for the reduced number of iterations is 0.0003 (df=24 and t=4.00). Therefore, our conclusion of COVAMOF resulting in a reduced number of iterations is statistically significant.

Finally, generalizeability is addressed by the fact that experiments in the ConIPF project (Configuration of Industrial Product Families, Hotz et al., 2006) with parts of the variability modeling concepts showed similar results. The experiment furthermore involved a realistic experiment in terms of size and complexity, and was based on an existing industrial product family. This product family was developed by an organization that faces the same issues as the organizations in the ConIPF projects, but that was not part of the case studies that were used to identify the issues that COVAMOF solves.

13.2.4. Q5 Variability Assessment Technique

The threats to the validity of the conclusions regarding this question are also related to correctness and generalizeability. COSVAM is an expert-based methodology that is strongly dependant on competences and specific situations. Moreover, applying COSVAM changes those situations. This affects repeatability of case studies, which makes it hard to proof correctness. The only way to overcome this issue is by continuously showing the benefits of COSVAM in multiple case studies.

At this point, COSVAM has been validated in one case study, which focused on qualitative results. Based on the results of that study, and the fact that the organization continues to use COSVAM, we are confident that COSVAM is a methodology that helps engineers to make the right choices when evolving the variability of a product family. In terms of generalizeability to other organizations, however, the validity has to be verified in case studies that involve other organizations as well. As we will show in the future work below, COSVAM thus represents a first large step into an area where many research challenges still remain.

13.3. Future Work

There are always points where a method can be improved or extended. The situation is not different for our research. In addition to the research directions we discussed in Chapter 5, we have considered numerous improvements and evolutions of COVAMOF that we were unable to realize within the time span of a Ph. D. track. We may or may not be able to realize those improvements in the future. By listing some of these improvements, we hope to inspire other researchers, both from the existing and new generation.

- As we discussed in Part III, several aspects remain that need to be validated in order to show the true benefit of applying COVAMOF for product derivation. For example, with the experiment we have shown that creating a COVAMOF model and subsequently deriving products is profitable, if the product family does not evolve. Whether applying COVAMOF is profitable in the long term is part of subsequent experiments.

- First, a number of Steps in COSVAM were performed manually, while some could have been supported by tooling. While our modeling tool COVAMOF-VS is capable of modeling variability, and finding variability mismatches, the COVAMOF-VS tool requires a number of extensions, such as maintaining the mismatch lists, and impacts of solutions scenarios, before our assessment method is fully integrated in the COVAMOF-VS tool-suite.
- In addition, we intend to investigate and incorporate existing quantitative and qualitative models for weighing the pros and cons of different solutions scenarios in COSVAM. With a stronger connection between our method and other of techniques, we think we can achieve a situation where also the weighing, scheduling, and selection of solutions can be supported by tooling.
- So far, we have also only briefly discussed the inaccuracy of provided variability in COSVAM. In case the assessment involves future products, e.g. during release planning, the required variability is also subject to inaccuracy, however. Predicted features may not be needed, for example, or new products may require a different set of features than predicted. We are currently extending the COSVAM to deal with inaccuracy of predictions, as well as ways to quantify the inaccuracy of provided and required variability.
- Applying COSVAM on multiple cases studies will not only further validate COSVAM, but also will enable us to extend it with a body of best practices, for example on the construction of product scenarios, the required level of detail of impacts of solution scenarios, a ranking of realization mechanisms with respect to common mismatches, and a collection of important goals and constraints with sensible weights.
- The COVAMOF tool-suite can be expanded so that the test results of a product that was configured during an iteration of the derivation process, can be automatically stored as Reference Data in the COVAMOF variability model.
- The way that the use of the COVAMOF tool-suite is depicted in this thesis is a situation where it is used to assist engineers during product derivation, i.e. through consistency checking, configuration guidance and automatic inference. This automatic inference is currently based on formalized knowledge. A COVAMOF model also contains documented knowledge, which consists of more fuzzy rules or hints regarding a general direction in which a configuration should be changed. In this situation, the product derivation process is still an iterative process where an application engineer is involved, as the engineer is needed to infer choices based on those hints. We envision a future where the role of the application engineer is taken over by a software tool, viz. a tool that also uses the fuzzy rules in the COVAMOF model and the test results of each iteration to derive a product autonomically.
- The concept of a software tool that autonomically configures a product is strongly related to runtime adaptability of software systems. We have suggested to incorporate the COVAMOF variability modeling elements as first

class entity in a programming language. In this way, for example, dependencies can be checked on runtime, and the system can be reconfigured using variation points. An article where this idea has been roughly outlined is (Siljee et al., 2005).

- Software product families is not the only research topic where choices and dependencies are relevant. In the past, for example, we have briefly investigated the relation to other topics, such as Model Driver Architecture (Deelstra et al., 2003) and Design Decisions (Sinnema et al., 2006). To reap the benefits of the research that has been performed on those topics and the research we present in this thesis, this relation could be elaborated more thoroughly.

13.4. Closing remarks

That is it. We have reached the final stage of filling this otherwise fine printing paper. This thesis is the result of large doses of heated discussions, coffee, humor, soup, inspiring work of fellow researchers, guidance and encouragements from our supervisors, loads of freedom to explore our own ideas, and an intense cooperation with industry. This last aspect is one of the key contributing factors to the success of our thesis. We are proud to see our results still being used outside of the academic world. We hope you enjoyed reading this thesis, and encourage you to take up and apply the ideas we presented here in your own organization, research, etc.

13.5. References

Bengtsson, P.O., Lassing, N., Bosch, J., van Vliet, H., 2004, “Architecture-level Modifiability Analysis (ALMA)”, Journal of Systems and Software, Vol. 69(1-2), pp. 129-147.

Clements, P., Kazman, R., Klein, M., 2001. Evaluating Software Architectures, Methods and Case Studies, Addison-Wesley, ISBN 0-201-70482-X.

Deelstra, S., Sinnema, M., van Gurp, J., Bosch, J., 2003. Model Driven Architecture as Approach to Manage Variability in Software Product Families, Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications (MDAFA 2003), CTIT Technical Report TR-CTIT-03-27, University of Twente, pp. 109-114.

Hotz, L. , Krebs, T., Wolter, K., Nijhuis, J., Deelstra, S., Sinnema, M., MacGregor, J., 2006. Configuration in Industrial Product Families - The ConIPF Methodology, IOS Press, ISBN 1-58603-641-6.

Siljee, J., Bosloper, I., Nijhuis, J., Hammer, D., 2005. DySOA: making Service Systems Self-Adaptive, Proceedings of The Third International Conference on Service Oriented Computing (ICSOC05), pp. 255-268.

Sinnema, M., van der Ven, J. S., Deelstra, S., 2006. Using Variability Modeling Principles to Capture Architectural Knowledge, Proceedings of the Workshop on SHAring and Reusing architectural Knowledge (SHARK2006).