ERAUSRAGENDE MASTERARBEITEN AM SC

FACHBEREICH Science & Engineering

STUDIENGANG Software Engineering for Embedded Systems

ASTERARBEIT

Feasibility Study on Variability Realization Methods in SysML Models

AUTOR/ N

Florian Rohlf

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

DISTANCE AND INDEPENDENT STUDIES CENTER

ABSTRACT

Model-based Systems Engineering (MBSE) has established itself as a successful approach to realize increasingly complex systems within an acceptable timeframe. However, rapidly changing and evolving systems as well as their growing distributed development pose additional challenges, especially with regard to the modifiability, adaptability and reusability of their components. In addition, the demand for highly flexible and customizable systems continues to grow. This results in a significantly greater need for an efficient variant management.

Proven approaches and methods already exist in the respective development disciplines to face these challenges. A solid MBSE approach, however, must provide a system-wide solution and answer how concurrent changes in a system model can be handled efficiently, especially if several similar system variants are developed in parallel. Industrial practice still shows a great deal of uncertainty in this respect. There are no conclusive answers to many questions. How can changes in a SysML model best be supported and, in particular, transferred effectively between model variants and versions? Should one model contain all configurations or is a separate variability model more useful? Which strategies are best suited to avoid imminent discrepancies between variant configuration and implementation and how can individual model components be efficiently reused?

In order to address these questions and provide practitioners with a helpful guideline, this master's thesis examines and compares existing approaches for realizing model variants in SysML with regard to their functionality as well as their effects (positive and negative) on the overall system concept. Since the focus lies on the feasibility of the shown approaches, they are applied by means of typical evolution scenarios and subsequently evaluated with regard to relevant performance indicators such as understandability, effort, granularity and independence. It is not expected that one approach is the best choice for every initial situation and under all circumstances. The introduced evaluation system thus aims to serve on the one hand as a situational decision support and on the other hand to offer the opportunity to examine, classify and evaluate own approaches and procedures more thoroughly.

KEYWORDS: Variability Management, Variability Realization, Model-based Systems Engineering (MBSE), Systems Modeling Language (SysML)

ZUSAMMENFASSUNG

Modellbasiertes Systems Engineering (MBSE) hat sich als erfolgreicher Ansatz etabliert, um immer komplexere Systeme in einem akzeptablen Zeitrahmen realisieren zu können. Sich schnell ändernde und erweiternde Systeme sowie deren zunehmend verteilte Entwicklung stellen jedoch große Herausforderungen dar, insbesondere hinsichtlich der Modifizierbarkeit, Anpassungsfähigkeit und Wiederverwendbarkeit ihrer Komponenten. Die Nachfrage nach hochflexiblen und anpassbaren Systemen steigt hingegen unentwegt an. Daraus ergibt sich ein hoher Bedarf an einem effizienten Variantenmanagement.

In den jeweiligen Entwicklungsdisziplinen gibt es bereits bewährte Ansätze und Methoden, um diesen Herausforderungen zu begegnen. Ein solider MBSE-Ansatz muss jedoch eine systemweite Lösung bieten und beantworten, wie gleichzeitige Änderungen in einem Systemmodell effizient gehandhabt werden können, insbesondere wenn mehrere ähnliche Systemvarianten parallel entwickelt werden. Die industrielle Praxis zeigt hier noch große Unsicherheiten. Auf viele Fragen gibt es keine schlüssigen Antworten. Wie können Änderungen in einem SysML Modell am besten unterstützt und insbesondere zwischen Modellvarianten und Versionen effektiv übertragen werden? Sollte ein Modell alle Konfigurationen enthalten oder ist ein separates Variabilitätsmodell sinnvoller? Welche Strategien sind am besten geeignet, um drohende Abweichungen zwischen Variantenkonfiguration und Implementierung zu vermeiden und wie können einzelne Modellkomponenten möglichst effizient wiederverwendet werden?

Um diesen Fragen nachzugehen und Anwendern einen hilfreichen Orientierungsrahmen zu geben, werden in dieser Masterarbeit bestehende Ansätze zur Realisierung von Modellvarianten in SysML hinsichtlich ihrer Funktionalität sowie ihrer Auswirkungen (positiv und negativ) auf das Gesamtsystemkonzept untersucht und verglichen. Da der Fokus auf der Machbarkeit der gezeigten Ansätze liegt, werden diese anhand typischer Evolutionsszenarien zunächst angewendet und anschließend hinsichtlich relevanter Leistungsindikatoren wie Verständlichkeit, Aufwand, Granularität und Unabhängigkeit bewertet. Es wird nicht erwartet, dass ein einzelner Ansatz für jede Ausgangssituation und unter allen Umständen die beste Wahl sein kann. Das eingeführte Bewertungssystem soll daher einerseits als situative Entscheidungshilfe dienen und andererseits die Möglichkeit bieten, eigene Ansätze und Vorgehen genauer hinterfragen, einordnen und bewerten zu können.

SCHLÜSSELWÖRTER: Variabilitätsmanagement, Realisierung von Variabilität, Modellbasiertes Systems Engineering (MBSE), Systems Modeling Language (SysML)

To get the full value of joy you must have someone to divide it with.

— Mark Twain

ACKNOWLEDGMENTS

Many thanks to everybody who supported me during this project. Writing a thesis is always accompanied by some time constraints and missed opportunities. I thank my friends and family for their understanding and continued support.

Special thanks go to Dr.-Ing. Martin Becker and Andreas Schäfer from the Institute for Experimental Software Engineering (IESE), who both have always been very supportive in all aspects of this undertaking. Without their comments, ideas, suggestions and discussions, this thesis could not have been realized.

As for typography, many thanks to Prof. Dr.-Ing. André Miede[1], who kindly provides a beautiful LaTeX-template which, despite some modifications and adjustments, servers as the basis of this thesis. A big thanks in this context also to the whole LaTeX-community for further support, ideas and simply great software.

1 https://www.miede.de/

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

API	Application Programming Interface
COTS	commercial off-the-shelf
CVL	Common Variability Language
DSL	Domain Specific Language
EA	Enterprise Architect
FODA	Feature-Oriented Domain Analysis
FPGA	Field Programmable Gate Array
IESE	Institute for Experimental Software Engineering
INCOSE	International Council on Systems Engineering
KPI	Key Performance Indicator
MBSE	Model-based Systems Engineering
MOF	Meta Object Facility
OMG	Object Management Group
OVM	Orthogonal Variant Model
PLE	Product Line Engineering
p::v	pure::variants
SPL	Software Product Line
SysML	Systems Modeling Language
UML	Unified Modeling Language
VAMOS	Variant Modeling with SysML
VCS	Version Control System
VEL	Variability Exchange Language
XML	Extensible Markup Language

1 INTRODUCTION

1.1 MOTIVATION

This thesis was written in cooperation with the Fraunhofer Institute for Experimental Software Engineering (IESE) in Kaiserslautern, Germany[1]. Motivated by their experiences from different industrial settings, it addresses the increasing need for guidance and decision support on how to handle variants and variability in SysML models in an adequate manner. While a substantial amount of variability realization approaches have already been discussed on the level of source code, there is little guidance for practitioners on the model level. With this, there is also major uncertainty in dealing with concurrent changes and parallel modeling of similar system variants. Although models are expected to cope with complexity, it can be observed that they sometimes become overly complex themselves due to poor modularization and variability realization.

The pressure of shorter development cycles and a stronger focus on individual, customer-specific requirements is causing the demand for reusable artifacts to rise steadily. In order to keep pace with these needs, the trend in systems engineering is clearly continuing towards Product Line Engineering (PLE). While this entails great methods to increase reusability and reduce time to market, it also introduces additional challenges, especially in terms of variability. Model-based Systems Engineering (MBSE) helps to increase system understanding and reduce overall complexity. However, in order to be equally successful in coping with variability, methods and mechanisms are required to deal with it systematically.

1.2 PROBLEM STATEMENT

While there are already a number of established methods to adequately realize variability on the level of source code [Pat11][Ape+13][ZDB16], there are hardly any concepts or studies on which approaches are suitable for a system-wide application, especially for models.

In order to bridge this gap and to give practitioners guidance for a suitable approach, this thesis will assess mechanisms for variability realization with respect to their feasibility for use with SysML. For this purpose, it is necessary to first give an overview of state-of-the-art approaches to variability realization from different development disciplines and to characterize their underlying principles/mechanisms. Finally, a subset of these mechanisms is transferred to an exemplary SysML model. In particular, product lines and their increased reusability requirements are also taken into account.

1 https://www.iese.fraunhofer.de

1.3 RESEARCH QUESTIONS

The following research questions (RQ) can be derived from the identified problem and shall be discussed within the scope of this thesis:

RQ1 Which mechanisms of variant realization exist and are suitable for system modeling with SysML?

RQ2 How can these mechanisms be assessed?

RQ3 Are there limiting factors to consider when choosing a mechanism?

A reference to these questions is given at the corresponding places in the thesis where they are discussed in more detail.

1.4 RESEARCH APPROACH

An overview of the main research activities is shown in Figure 1.1. A large part of these activities (framed in red) is devoted to compiling the results of related work and defining the conceptual model. Adjacent topics of interest are also outlined to help narrow down and clarify the main subject, the realization of variability. The expected results (output) and the main resources used (input) are also briefly listed. The representation corresponds essentially with the structure of this thesis.

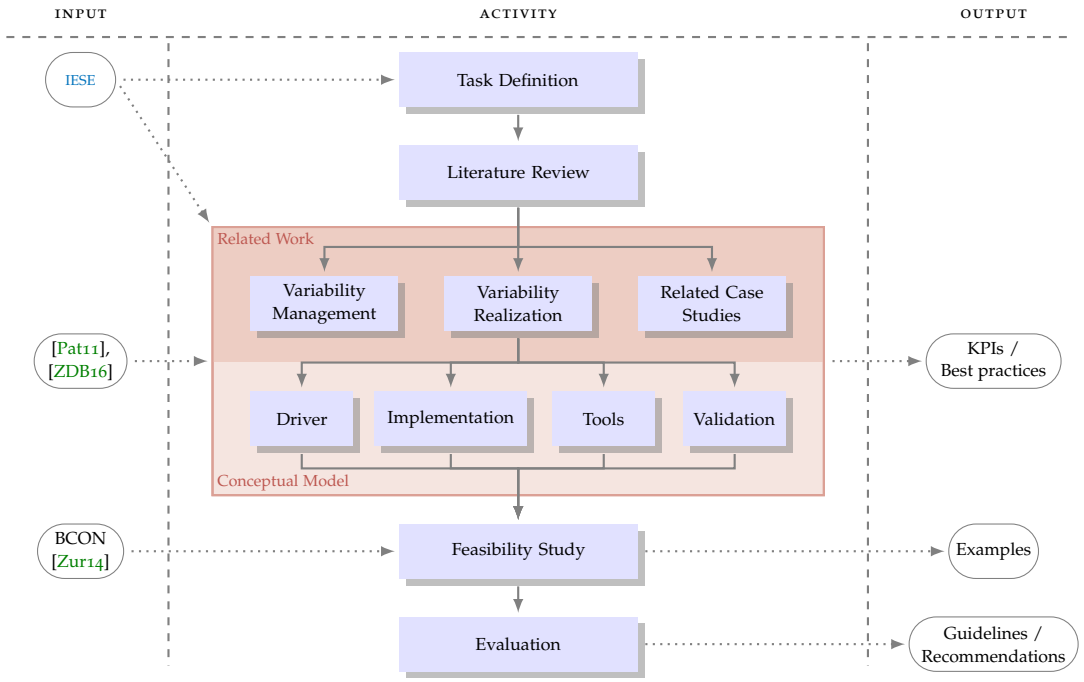


Figure 1.1: Overview of methodical research process

1.5 MAIN CONTRIBUTIONS

As Figure 1.1 indicates, the thesis is intended to make the following contributions:

- Summarize the current state of research regarding methods and mechanisms to realize variability.
- Provide examples that demonstrate the feasibility of these mechanisms within a SysML model.
- Establish guidelines and recommendations for dealing with variability in the MBSE context, derived from the examples provided.

1.6 THESIS STRUCTURE

The remainder of this thesis is structured as follows. Chapter 2 describes the foundation. All important concepts and methods on which the following chapters are based are introduced here. A reference to current research results is intended to provide a comprehensive overview. Section 2.3 elaborates on the literature review carried out for this purpose.

Chapter 3 introduces the conceptual model of the thesis and describes various aspects of variability. At first, possible causes and triggers for variants are introduced in Section 3.1. Section 3.2 discusses the possibilities and challenges of different variant management approaches, followed by a more detailed examination of the required tool support in Section 3.3. The main part is presented in Section 3.4, where mechanisms for the realization of variability and its underlying principles are described. A list and explanation of the Key Performance Indicators (KPIs) that can be used to evaluate the different mechanisms concludes the chapter.

In Chapter 4, an exemplary product line evolution is introduced first. Its changes are successively implemented in a SysML model to demonstrate the feasibility of the methods and mechanisms shown. For this purpose a model of the virtual company BCON is used [Zur14]. An evaluation of the implementation based on the presented evaluation criteria concludes the feasibility study.

Chapter 5 summarizes the results of the thesis and offers an outlook on further applications.

The Appendix contains information that did not into the body of this work for reasons of better readability.

2

FOUNDATION

In this chapter the basic concepts and terms relevant and necessary for the thesis are introduced and explained. Furthermore, it is intended to give an overview of existing publications on these subjects.

2.1 ENGINEERING APPROACHES

Systems engineering became popular in the 1940s as a means of dealing with emerging problems of increasingly complex systems. Initially driven by the rapidly growing telephone industry, aerospace soon took over the leading role. Methods were sought to cope not only with the increasing complexity, but also with the greater interdisciplinarity and growing importance of component interaction. Important tasks are the elaboration of a consistent system specification and requirements analysis, quality assurance of all components and their system integration, risk and configuration management and system validation. Systems engineering is intended to organize all the different disciplines, provide methodical support in solving problems and maintain a holistic view of the system. The International Council on Systems Engineering (INCOSE) is a non-profit membership organization and professional association in the field of systems engineering. The following definition is taken from their handbook:

Systems engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, and then proceeding with design synthesis and system validation while considering the complete problem: operations, cost and schedule, performance, training and support, test, manufacturing, and disposal. [INC15, p. 11]

Especially with the "increasing complexity of systems" and the growing number of interacting engineering disciplines, "there is a rising need for interdisciplinary systems engineering" [BZ18]. Nowadays, almost every company uses systems engineering, albeit in different depths and with different names. The tasks and methods often overlap with those of product and project management. In software-intensive projects, the function is sometimes assigned to software architects who have the necessary knowledge of the system topology.

2.1.1 MBSE CONCEPTS

A traditional way used by scientists to master the increasing complexity and variability of real-world phenomena is to resort to modeling [Jéz12]. Model-based Systems Engineering (MBSE) can therefore be considered as an advancement of systems engineering at a higher level of abstraction. But more than that, it applies the well-proven engineering concept of divide and conquer. A complex system can be broken down into as many models as needed in order to address all the relevant concerns in such a

way that they become understandable enough. The models itself may be expressed with any appropriate modeling language [Jéz12].

Initially, a model describing the rough system structure is established, which is then refined in further development steps. All results from the respective development steps are described and recorded within this model, to ensure a consistent and valid data basis. The system description distinguishes between two system models: structural and behavioral models. Structural models are used to specify the structural design of the system and the technical realization of the interaction capability of system elements. Behavioral models are used to define the system behavior and the type of communication that the structural elements use [Sch19].

Figure 2.1 depicts the three important concerns in the context of system modeling: Method, Tool and Language. Ideally, they all work hand in hand and thus support the process of developing a sound system model.

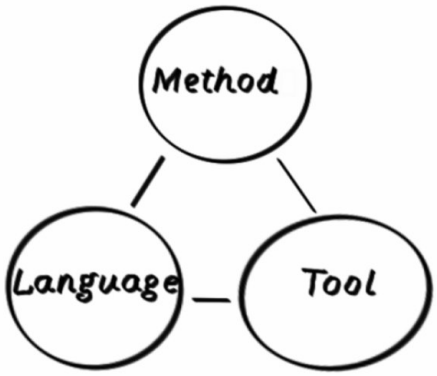


Figure 2.1: Three concerns of systems modeling [Wei16, p. 7]

In practice, however, systems engineers often face many constraints in the selection process. Tools are often very expensive, and their integration into a company can be very time-consuming because the usage has to be well trained. The acceptance rate for changes/innovations is sometimes low, especially in more traditional companies. In addition, already established processes can lead to further limitations in the choice of methods and languages. Some examples for each of these three concerns are:

Method: SYSMOD[1], VAMOS[2], OOSEM[3], RUP SE[4], ...

Tool: Enterprise Architect, Cameo Systems Modeler, objeciF RM, IBM Rhapsody, ...

Language: SysML, UML, BPMN[5], ...

Across all application domains, the interest in MBSE has increased significantly in the last decade. While MBSE mainly has been followed by early adopters 10 years ago, meanwhile the extensive usage of SysML/UML models has become mainstream.

1 The Systems Modeling Toolbox
2 Variant Modeling with SysML
3 Object-Oriented Systems Engineering Method
4 Rational Unified Process for Systems Engineering
5 Business Process Model and Notation

Experiences from past and current projects of the Institute for Experimental Software Engineering (IESE) with industry partners as well as statements from a survey on systems engineering [Hei+17] and other industry publications resulted in the identification of the following driving forces that motivate companies to adopt MBSE approaches:

Increasing complexity. The increasing complexity of products, accompanied by increasing interdisciplinarity with shorter change cycles, makes it necessary to use processes and methods that counteract this. MBSE supports the satisfaction of the resulting increasing need for cooperation and technical understanding between the different disciplines. This creates a basis for discussing complex, interdisciplinary problems and reduces the risk of misunderstandings caused by different definitions of terms (cf. [ERZ14][Kai14][INC15]).

Standard adherence. Customers oblige their suppliers to assure process quality on a given maturity level, e.g. by adhering to some process maturity model. This is for instance the case in the automotive domain. Automotive SPICE[6] is a domain-specific variant of the international standard ISO/IEC 15504. The purpose of Automotive SPICE is to assess the performance of the development processes of ECU[7] suppliers in the automotive industry. The standard requires suppliers to engineer and use respective system and software architecture artifacts in the course of the engineering. Being extrinsically motivated, many companies seek for approaches to provide the necessary models in a cost-efficient manner. Hence, they are highly interested in reuse of model parts.

Tool availability. The availability of promising modeling and analysis tools supports the establishment of engineering and manufacturing networks. This supports the reduction of the manufacturing and development depth and allows the development to become increasingly specialized. Also, the activity of coordinating the engineering projects moves further into the foreground of an engineer, similar to the activity of a manager (cf. [FG13]).

Higher abstraction level. Last but not least, the representation of systems at a higher level of abstraction is made possible and a platform is created for discussion with the various stakeholders, some of whom may not even be familiar with the topic. This promotes a common understanding and offers the possibility to identify ambiguities early. The representation of a cross-system communication can in some cases be more easily understood by three diagrams than by 80 pages of specifications. It will also greatly facilitate and improve communication between all the actors involved. In addition to increasing comprehensibility, an increase in the degree of abstraction can support the implementation of software by using suitable models. Models can be used to specify the behavior and automatically derive software implementations or verification and validation artifacts (cf. [Kai14][INC15]). This is similar to the code level and its evolution from assembler to higher, more abstract programming languages and its associated increase in comprehensibility and reduced development time.

6 Software Process Improvement and Capability Determination
7 Electronic Control Unit

The above criteria show some general motivations for the introduction of MBSE. However, the realization and especially the degree of realization can vary greatly. In its simplest form, MBSE can be used to improve communication and comprehensibility. In a next step the aim would be to establish a system-wide traceability. Meaning, that the stakeholder needs can be consistently traced to their implementation and are also used for system verification. This consistency implies a greater level of detail and increases the interdependencies between diagrams (model views). In a final step, the model itself becomes the system specification. Consequently, it not only adds information to any existing system documentation, but itself represents all aspects of the system. Required documentation can thus be derived from it as needed.

As shown in Figure 2.1, systems modeling is not bound to any particular language. In the context of this thesis, however, SysML was chosen for this purpose because it is widely used and offers good tool support [AZ13]. SysML is a graphical, Unified Modeling Language (UML) based, standardized modeling language. The standard was published in 2007 as a joint effort of the Object Management Group (OMG) and INCOSE and is currently released under version 1.5. Within SysML there are a variety of different diagrams which can be (with the exception of the requirements diagram) divided into two categories, structure and behavior. The following chapters assume a basic familiarity with SysML, its diagram types and their usage. Additional information can be found in [Wei14]. As far as the other two aspects (tools and methodology) are concerned, initially no stipulation is made in order to be as widely applicable as possible. For the feasibility study, however, Enterprise Architect (EA)[8] will be used as the tool for implementation of the system model and its changes.

2.1.2 PLE CONCEPTS

Product Line Engineering (PLE) is a way to engineer a portfolio of related products in an efficient manner, taking full advantage of the products' similarities while respecting and managing their differences. "Engineer" in this context means all of the activities involved in planning, producing, delivering, and deploying, sustaining, and retiring products [YC17].

To this end, PLE separates the engineering process into the two major activities of application and domain engineering. Domain engineering concentrates on all common aspects of a product line and provides reusable parts called *domain assets*. An important part of this is scoping, which continuously checks whether components should be implemented as domain assets, and thereby enables strategically planned reuse [Bec17]. Application engineering on the other hand, creates products by using domain assets as well as creating new (application specific) assets. Scoping and domain engineering focus therefore on a development for reuse, while application engineering implements a development with reuse. Another aspect, that can be regarded as a high-level concept, is the separation of problem and solution space. Any product, that is actually build or intended for use by a customer exists in the solution space. The problem space contains all the customer needs that a product shall be able to fulfill. In other words, the product in the solution space implements the hypotheses of the developer on how to meet the customer needs from the problem space. Complaints, requirements and

8 https://www.sparxsystems.de/

changes can often only be formulated in the problem space and need to be transferred to the solution space in order to improve the product hypothesis [Ols15].

The four-part division resulting from the combination of the problem space and solution space with domain and application engineering is shown Figure 2.2. The problem space describes hereby the variable and common parts of the product line, reflecting the desired range of product variants. The associated solution space describes the constituent assets of the product line and its relation to the problem space, i.e. rules for how elements are selected. Several different options are available for modeling the information in these four quadrants. The problem space can be described e.g. with Feature Models, or with a Domain Specific Language (DSL). There are also a number of different options for modeling the solution space, for example component libraries, DSL compilers, generative programs and also configuration files [pur20].

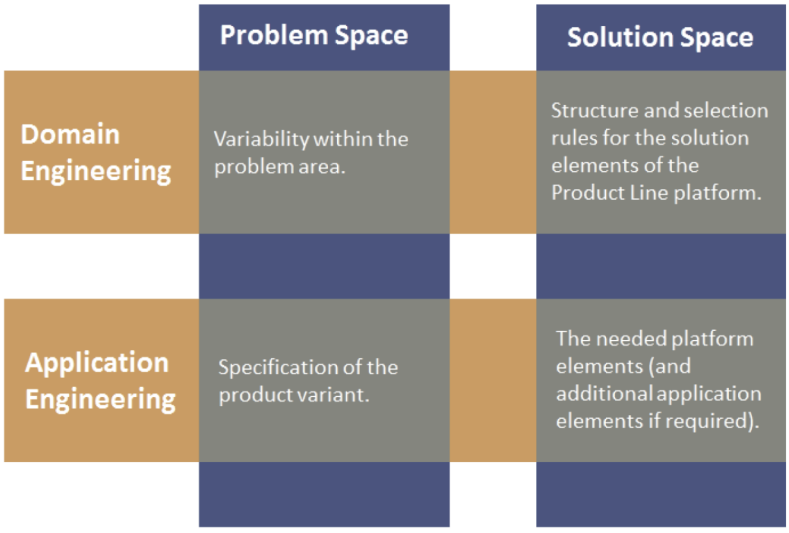


Figure 2.2: Variability in the PLE context [pur20, p. 6]

Variability in the PLE context must therefore be viewed from two perspectives. The first one being the variability in the problem space where a specific product is decomposed as a set of features that can appear within that product. The features in the problem space represent what functionality must be implemented in the system. On the other hand, solution space variability represents the variability within design artifacts which are used to extend, customize or change the system in order to implement product features. In other words, the solution space represents how the features are implemented in the given system [Bil+18].

The beneficiaries of these two representations are not identical. While product features and characteristics are described from the perspective of system users/designers, the solution space focuses on developers, who are provided with a selection of implementation variants. Variation in the problem space automatically lead to at least one corresponding variation in the solution space. This does not necessarily apply vice versa. Therefore, a larger amount of variation points in the solution space can be assumed. Interestingly, as the literature review will show, significantly more approaches and publications deal with the presentation and management of variants rather than with their realization. Both aspects will be further elaborated in Sections 2.4 and 2.5.

2.2 VARIABILITY CONCEPTS

Variability represents a capability to change or adapt a system and has been recognized as the key to systematic and successful reuse [Bec03]. As mentioned before there are two aspects to consider, especially in the context of PLE. A variability at the specification level (problem space) describes a design decision that can affect the behavior of the system as well as its qualities. A variation point at the implementation level (solution space) will implement this variability by using appropriate mechanisms. Variation points therefore represent variability, as a spot in an asset where variation will occur, i.e. where variability is realized [Bec03].

In practice, different approaches for reuse can be considered when applying product line engineering. These differ in type and extent of reuse. Figure 2.3 shows an overview of practical reuse approaches. With regard to planning effort, a distinction is made between ad-hoc and strategic approaches. A further distinction concerns the artifacts that arise. While cloning creates new and independent assets, the other approaches aim at developing a common platform with a higher degree of reusable parts.

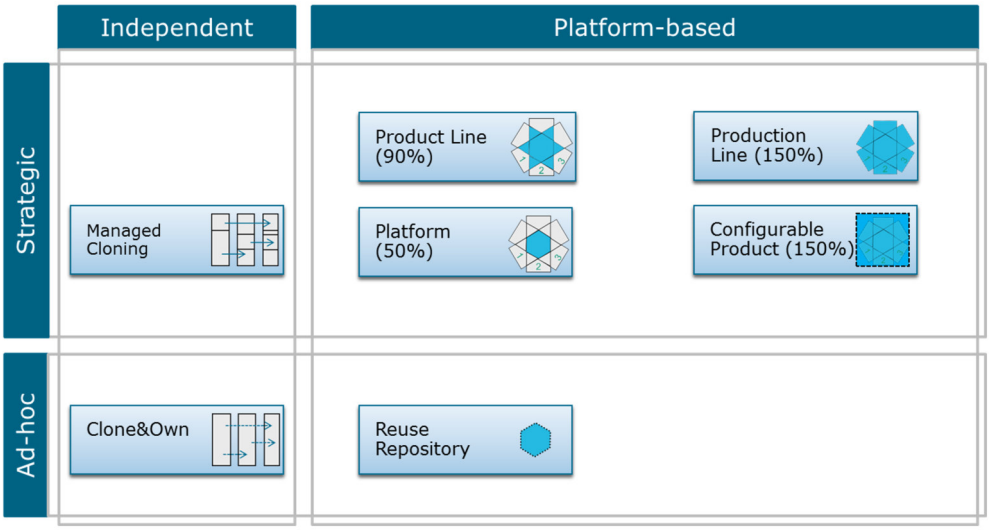


Figure 2.3: PLE reuse approaches [Bec17, p. 32]

Clone & Own. Copy and modify can be applied to any artifact. It requires no planning. The approach will be explained in more detail in Section 3.4 as a basic mechanism for realizing variability.

Reuse Repository. The creation of a repository from which parts can be selected and reused if needed. In contrast to the platform-based approaches, the repository is not a result of deliberate planning. Rather, it is a collection of artifacts that could potentially be reused in the future. Artifacts in the repository are therefore not necessarily designed for reuse and may need modifications before their application.

Managed Cloning. Unlike the spontaneous Clone & Own, managed cloning attempts to manage and strategically plan reuse. However, there are still no shared assets – they remain to be created by copying – and therefore no variation points exist within the artifacts.

Platform. Reusable components are stored as domain assets and form the common basis for all products. Only assets used in all products are considered part of the platform. Therefore, no variability exists within the domain assets. All specialization is implemented via application assets.

Product Line. By extending the platform to also include specialized assets that are not used by all but only some products, variation points are introduced in the domain assets. This increases the proportion of potentially reusable components of the common platform.

Production Line. In a production line, the common platform integrates all assets. There is no application engineering anymore. The generation of products is done by selecting the corresponding domain assets.

Configurable Product. A product that contains all variants and takes on its specific form through configuration (at runtime). Variability exists only in the domain assets [Bec17].

There is no general statement as to which approach is most suitable. The decision depends on many factors, e.g. number of products/variants, development history, commonalities, etc. Section 3.2 will elaborate on this further and examine the effects of a chosen approach on the realization of variants in more detail.

At the implementation level, Patzke defines a variability mechanism as "a particular way of intentionally realizing variability in core assets. The purpose of variability mechanisms is to balance reuse effort and evolution effort by efficiently organizing common elements and variants, as appropriate in the particular context of product line engineering" [Pat11, p. 46]. This means a specification describing how to specialize the variation point to a distinct variant, and a mechanism to realize this specialization is required. A single point of variation is thus associated with a mechanism that handles the variability. Various mechanisms can be used for this purpose. These mechanisms can be roughly categorized into three classes. [Bec03]:

Selection. An existing solution can be selected to specialize the variation point. The selection is described in the corresponding specification.

Generation. The generation of a solution, e.g. through an external generator. The specification forms the input of the generator and the generated output specializes the variation point.

Substitution. Supports the specialization of the variation points by unique, externally provided solutions. Therefore, the corresponding variation points can be considered as some kind of interface.

Sections 2.5.1 and 3.4 will further investigate and explain concrete mechanisms of all these three classes, particularly relevant for model-based systems engineering.

2.2.1 TYPES OF VARIABILITY

The type of variability can be related to what can be expressed by the variation point i. e. what the variation point can be used for. It is important to note that variation points can affect both the architecture and the element level. In other words, at the element level they refer to what elements can be attached to the variation points. But from an architectural point of view, setting the variation points themselves can also be used to represent variability and can be understood as an expression of architectural/topological variability. The following list presents the three types of variability:

(1) Option
    a) Optional

(2) Selection
    a) Multiplicity
        i. Alternative
        ii. Multiple coexisting
    b) Openness
        i. closed
        ii. open range

(3) Topology

Option. An option is the simplest form of a variation point. A feature constrained by the variation point can be either present (selected) or absent (not selected) in the resulting variant. There is no third choice.

Selection. In contrast to the option where a binary decision is required, the selection offers a choice between several alternatives. Two further aspects must be taken into account hereby: Multiplicity and Openness. Does the selection of one alternative exclude others or can multiple alternatives be selected concurrently? Is the list of possible alternatives complete and closed or can new alternatives be added? Especially at the beginning of a product line development it is often not possible to answer the second question conclusively. Nevertheless, it is important to consider at an early stage whether many changes can be expected at a variation point and additional variants are to be expected. During implementation, attention can already be paid to extensibility and the model/architecture can be prepared accordingly.

Topology. Another aspect to consider is the already mentioned topological variability. When choosing variation points it is necessary to be aware of how and where they will affect the model. In case a variation point is used several times in several different views and diagrams throughout the model, keeping an overview of its impact is much more difficult. Adding another alternative for such a variation point can therefore be both time-consuming and error-prone. It is important to note that this does not mean a high number of different variation points, but rather the effects of a single variation point being used at multiple locations in the model. This challenge can only be addressed by appropriate tool support and a well designed system architecture.

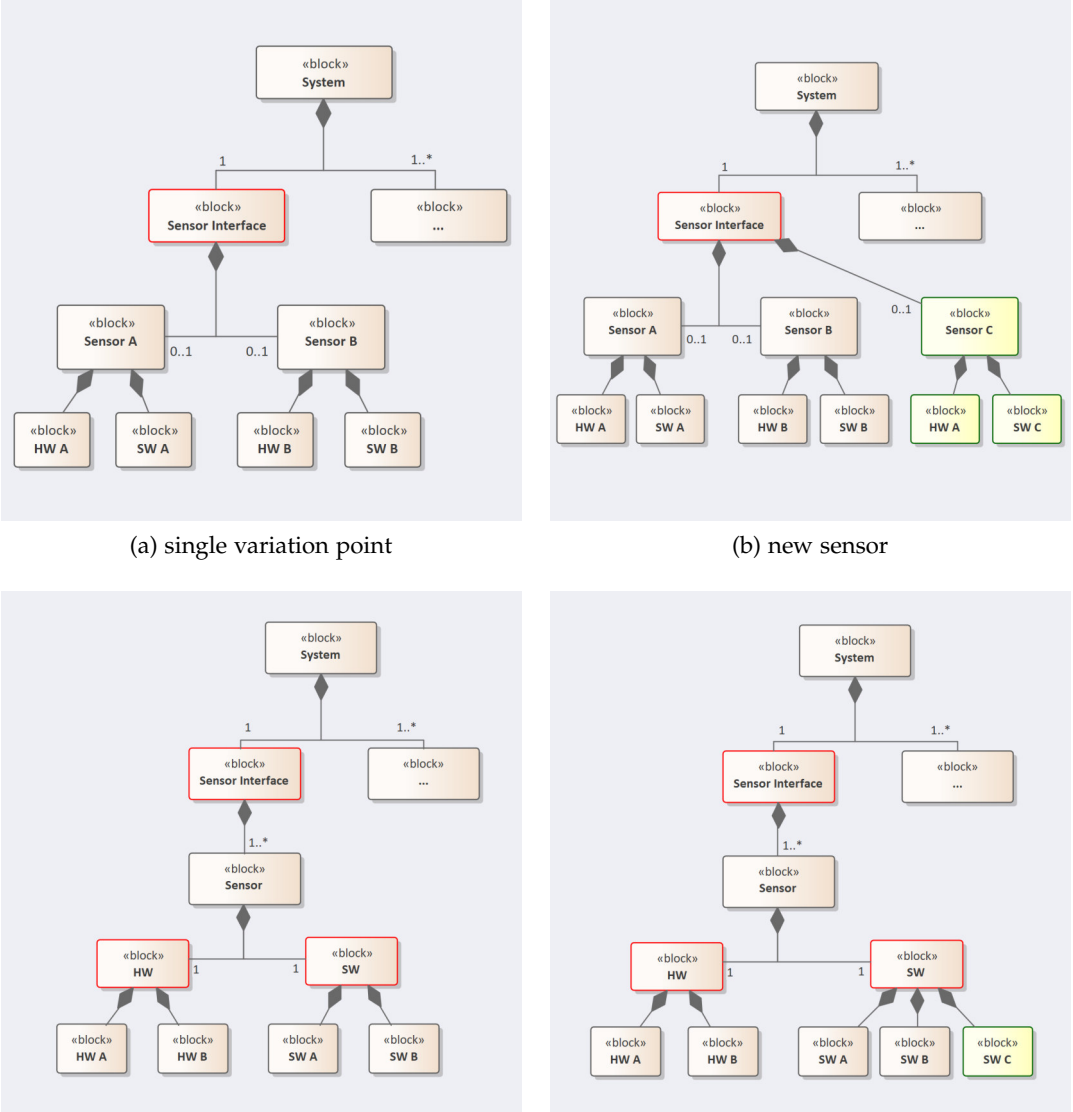
From a product development perspective, it can already be assumed that both variation points in the form of options and simple alternatives should not cause any major difficulties. In contrast, open and topological variability already show the potential to be the main complexity drivers in this respect.

2.2.2 LEVEL OF ABSTRACTION

In order to achieve high reusability of components, it is desirable to be able to insert variation points basically everywhere. The extent to which variation points can be inserted is merely limited by the level of detail of the system model and the technical possibilities offered by the tools used. As already mentioned, the level of detail of the system model can be freely chosen (depending on what is to be achieved with the model) and may even change over its life cycle. The selection of appropriate variation points is therefore not a trivial task, especially with regard to the resulting number of variants and the topological dependencies. Considering that the more variation points exist, the more ambiguous and complex the model can become. It is therefore a trade-off between higher reusability and higher complexity. This shall be illustrated by a simple example.

Figure 2.4 shows a system equipped with sensors that shall be connected via its sensor interface. For simplicity, a sensor itself consists of only one software and hardware component. Other components and details are not relevant at this point. Figure 2.4a shows the decision to introduce only a single variation point in the sensor interface. Elements that contain a variation point are outlined in red. For each variation of the interface, therefore, both the hardware and software aspects have to be modeled. Figure 2.4b now introduces a new interface variant (Sensor C) outlined in green, that consists of HW A and SW C. In Figure 2.4c, the same system is modeled, but more variation points have been added to the hardware and software components. In addition, the specific sensors were replaced by a new, generic element, which always consists of a hardware and a software component. Figure 2.4d shows the changes necessary to implement Sensor C in this setup.

This simple example already shows that the reuse of common parts increases with the number of variation points. In Figure 2.4b the redefinition of HW A is redundant. Nevertheless, the visibility of the individual variants in Figures 2.4c and 2.4d is significantly less, because a concrete instance of a sensor is not shown. Additional considerations and restrictions must be documented separately to show which combinations of hardware and software are allowed when selecting a particular variant and thus form a sensor. Immediate information about which and how many sensors are available is missing.



(a) single variation point

(b) new sensor

(c) multiple variation points

(d) new component

Figure 2.4: Level of abstraction

What is also evident from Figure 2.4 is a difference in architecture between realization with single and multiple variation points. At a higher level of abstraction, variation points are easier to manage and understand. The level of abstraction refers to the degree of complexity with which a system is viewed. The higher the level, the less detail. The lower the level, the more details. The highest level of abstraction is the entire system. The decomposition of the system into more and more components results in more possibilities for variation points and thus potentially greater interdependencies and complexity. It is therefore important to stress the relevance of deliberate planning as to which components can potentially contain variation points. If the architecture can be adapted to this at an early stage, it simplifies later changes tremendously.

2.3 LITERATURE REVIEW APPROACH

According to Fink a research literature review is "a systematic, explicit, [comprehensive, (p. 44)] and reproducible method for identifying, evaluating, and synthesizing the existing body of completed and recorded work produced by researchers, scholars, and practitioners" [Fin10, p. 3].

The procedures proposed by Fink and other studies respectively their underlying guidelines [KC07] were mostly adopted for the preparation of this literature review. One study “Variability in Software Systems - A Systematic Literature Review” is particularly noteworthy, as it was carried out in 2014 on a very similar objective [Gal+14]. Results of this study will be taken into account in the appropriate places.

Although an extensive and conclusive literature research as proposed by [Fin10] could not be carried out due to a lack of time, an overview of all activities conducted in this regard is given in this section. First the approach is described briefly. Then the used databases are named and all queries are listed. Finally, the results are shown and summarized with regard to the research questions formulated in Section 1.3. A more detailed list and evaluation of potentially relevant documents obtained by this method concludes the review of related work.

Approach. The search has mainly focused on automated searches within digitally accessible libraries. In addition to that, a manual target search with IESE internal resources was carried out (including documentation of best practices, project reports and developed approaches in the context of PLE, MBSE and Systems Engineering). Cross-references found in the literature were also checked for their relevance. In order to remain as up-to-date as possible, the search was repeated several times during the course of this thesis, paying particular attention to additions and changes. All statements refer to the result of the latest search on 2020-07-15. Special attention was given to publications and authors in connection with the two major conferences SPLC[9] and MODELS[10].

The following inclusion criteria were applied to the contents found:

- Papers matching the search string and describe approaches within the scope of this thesis.

In order to reduce the number of results to a reasonable level, the following exclusion criteria were applied:

- Papers not focusing on the realization of variability in the MBSE context.
- Papers not written in English or German.
- Papers not accessible through one of the stated databases or from IESE.

9 Systems and Software Product Line Conference
10 Model Driven Engineering Languages and Systems

Databases. The following search databases were used to identify relevant papers and publications. All of them are known to contain a large number of publications, especially in the field of (computer) science.

- Google Scholar (http://scholar.google.com/)
- Digital Bibliography & Library Project (https://dblp.uni-trier.de/)
- ScienceDirect Elsevier(http://www.sciencedirect.com/)
- ACM Digital Library (http://dl.acm.org/)
- IEEE Xplore (http://ieeexplore.ieee.org/)
- Semantic Scholar (https://www.semanticscholar.org/)

Since the searches were repeated several times, it was particularly noticeable that the results of *Semantic Scholar* increased steadily. Within only 6 months, the number of entries found for unchanged search queries increased to more than 30 times the original value. Unfortunately, the relevance of the found entries did not increase to the same extent.

Search strings. In accordance with the research questions (cf. 1.3), the following database search queries were used to identify relevant documents:

- "variability management" AND "MBSE"
- "variability management" AND "model-based systems engineering"
- "variability management" AND ("model-based systems engineering" OR "MBSE")
- "variability realization" AND ("model-based systems engineering" OR "MBSE")
- "variability realization" AND "sysml"
- ("variability realization" OR "variability management") AND ("model-based systems engineering" OR "MBSE") AND "sysml"

Search results. The total number of search results is shown in Table 2.1. The evaluation and extraction of relevant papers from these results can be found in Table 2.2 on pages 16 to 17. In general, the results can be classified into the two areas of variability modeling and variability realization. Since the term variability modeling is slightly misleading, as it is less about variability in the context of MBSE, but rather about visualizing and organizing variability capabilities (of a product/product family) as such. In the context of this thesis the term variability management is therefore used.

The already mentioned study [Gal+14] by Galster et al. shows that the number of published papers is highest in the two major development areas *design* and *architecture*. Especially in these areas, the focus is on the visualization aspect of feature variability and thus the variability management. In contrast, the realization of variability during the *implementation* phase is only addressed in about 10% of the reviewed studies. The obtained search results support this statement. Significantly more studies and publications could be found in connection with the modeling and structuring of variability within product families than for their realization.

SEARCH STRING	GOOGLE SCHOLAR	DBLP	SCIENCE DIRECT	ACM DL	IEEE XPLORE	SEMANTIC SCHOLAR
"variability management" AND "MBSE"	81	0	1	3	0	1240
"variability management" AND "model-based systems engineering"	112	1	8	4	1	1600
"variability management" AND ("model-based systems engineering" OR "MBSE")	133	1	8	4	1	1750
"variability realization" AND ("model-based systems engineering" OR "MBSE")	28	0	0	2	0	1210
"variability realization" AND "sysml"	38	0	2	2	0	1700
("variability realization" OR "variability management") AND ("model-based systems engineering" OR "MBSE") AND "sysml"	84	0	5	3	1	282

Table 2.1: Literature search results from 2020-07-15, 8:00 pm

Although the focus of this thesis is on the implementation aspect of variability, both areas and the approaches contained therein will be briefly described in the following sections. On one hand, this helps to distinguish their underlying mechanisms more clearly, and on the other hand, both aspects can often not be applied in complete isolation. Decisions in the choice of a management method have an impact on the possible implementation options.

TITLE	AUTHOR	PUBLISHED	COMMENT	RELEVANCY
Flexible Product Line Derivation Applied to a Model Based Systems Engineering Process	Cosmin Dumitrescu, Patrick Tessier, Camille Salinesi, Sebastien Gérard, Alain Dauron	"Complex Systems Design & Management 2012" pp. 227–239 Springer, 2013	▶ development of a flexible configuration process for product line deviation ▶ automotive context	○
Capturing variability in Model Based Systems Engineering	Cosmin Dumitrescu, Patrick Tessier, Camille Salinesi, Sebastien Gérard, Alain Dauron, Raul Mazo	"Complex Systems Design & Management 2013" pp. 125–139 Springer, 2014	▶ use of CO-OVM ▶ focus on the representation of variability ▶ automotive context	○
Coping with Variability in Model-Based Systems Engineering: An Experience in Green Energy	Salvador Trujillo, Jose Miguel Garate, Roberto Erick Lopez-Herrejon, Xabier Mendialdua, Albert Rosado, Alexander Egyed, Charles W. Krueger, Josune de Sosa	ECMFA 2010: "Modelling Foundations and Applications" pp. 293–304 [Tru+10]	▶ variability management and implementation with SysML ▶ over-exposed (150%) model ▶ energy context	◐
A review of know-how reuse with patterns in model-based systems engineering	Quentin Wu, David Gouyon, Éric Levrat, Sophie Boudau	"Complex Systems Design & Management 2018" pp. 219–229	▶ knowledge and know-how preservation and reuse through patterns	○
Bridging the gap between product lines and systems engineering: an experience in variability management for automotive model based systems engineering	Cosmin Dumitrescu, Raul Mazo, Camille Salinesi, Alain Dauron	SPLC 2013: Proceedings of the 17th International Software Product Line Conference, pp. 254–263	▶ use of CO-OVM ▶ focus on domain engineering ▶ automotive context	○
Model-based systems engineering with requirements variability for embedded real-time systems	Mole Li, Firat Batmaz, Lin Guan, Alan Grigg, Matthew Ingham, Peter Bull	Proceedings of 2015 5th IEEE International Model-Driven Requirements Engineering Work-shop (MoDRE) Canada, 2015, pp. 36–45	▶ variability modeling by extending OVM ▶ aerospace context	◐

continues on next page

TITLE	AUTHOR	PUBLISHED	COMMENT	RELEVANCY
How to Boost Product Line Engineering with MBSE - A Case Study of a Rolling Stock Product Line	Hugo G. Chalé Góngora, Marco Ferrogalini, Christophe Moreau	"Complex Systems Design & Management 2014" pp. 239–256	▶ PLE reuse strategies ▶ over-exposed (150%) model in combination with variability model (OVM) ▶ transportation context	◐
Towards Solving MBSE Adoption Challenges: The D3 MBSE Adoption Toolbox	Mohammad Chami, Aiste Aleksandraviciene, Aurelijus Morkevicius, Jean-Michel Bruel	INCOSE International Symposium Vol. 28, Issue 1, 2018, pp. 1463–1477	▶ MBSE adoption approach	○
Model-based Product Line Engineering – Enabling Product Families with Variants	Matthew Hause, James Hummell	IEEE Aerospace Conference, 2015	▶ over-exposed (150%) model in combination with variability model (OVM) ▶ aerospace context	◐
Model-based product line engineering in an industrial automotive context: an exploratory case study	Damir Bilic, Daniel Sundmark, Wasif Afzal, Peter Wallin, Adnan Causevic, Christoffer Amlinger	SPLC 18: Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 2 [Bil+18]	▶ variability management and implementation with SysML ▶ over-exposed (150%) model ▶ automotive context	●
CO-OVM: A Practical Approach to Systems Engineering Variability Modeling	Cosmin Dumitrescu	PhD Thesis, Université Panthéon-Sorbonne - Paris I, 2014 [Dum14]	▶ introduction of CO-OVM	◐
An Integrated Model-based Tool Chain for Managing Variability in Complex System Design	Damir Bilic, Etienne Brosse, Andrey Sadovykh, Dragos Truscan, Hugo Bruneliere, Uwe Ryssel	IEEE / ACM 22nd, International Conference on Model Driven Engineering Languages and Systems, 2019 [Bil+19]	▶ over-exposed (150%) model ▶ automated variability information exchange ▶ automotive context	●
Software Product Line Engineering and Variability Management: Achievements and Challenges	Andreas Metzger, Klaus Pohl	FOSE 2014: Proceedings of the on Future of Software Engineering, 2014, pp. 70–84 [MP14]	▶ summery of major research achievements in PLE and variability management	◐
Variability in Software Product Lines	Felix Bachmann, Paul C. Clements	Software Engineering Institute, Pittsburgh, USA, 2005	▶ guidelines for the creation of core assets with included variability	○
Variant Modeling with SysML	Tim Weilkiens	MBSE4U Booklet Series, 2016 [Wei16]	▶ VAMOS approach	◐
A survey of variability modeling in industrial practice	Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, Andrzej Wąsowski	Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, 2013 [Ber+13]	▶ industrial survey ▶ compares the use of different modeling approaches during various development phases	◐
Three Cases of Feature-Based Variability Modeling in Industry	Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, Andrzej Wąsowski	Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014 [Ber+14]	▶ exploratory case study about feature modeling ▶ industrial practices	◐
How Domain-Specific Modeling Languages Address Variability in Product Line Development	Juha-Pekka Tolvanen, Steven Kelly	Proceedings of the 23rd International Systems and Software Product Line Conference, 2019 [TK19]	▶ focus on domain specific languages	◐
● = high relevancy	◐ = medium relevancy	○ = low relevancy		

Table 2.2: Literature search result evaluation

2.4 VARIABILITY MANAGEMENT

This section presents approaches and concepts that focus on the modeling aspect of variability in the problem space. It is about visualizing product or feature variability and managing their dependencies. The presented approaches are notable because of their frequent mention in literature as well as their successful application in practice.

According to Metzger and Pohl [MP14], there are generally two approaches as to how the variability of a product line can be explicitly specified and represented in a model:

Integrated documentation. The integrated documentation describes all variation points and their characteristics directly inside the model. For this purpose, numerous modeling languages have been proposed, ranging from simple annotations to domain-specific language extensions.

Orthogonal documentation. The orthogonal documentation captures all variability aspects in a separate model. The resulting variability model does not contain any design decisions or feature descriptions. The strict separation aims to reduce overall complexity and increase comprehensibility. To ensure system consistency, the variability model relates the variability it defines to other (software) development models such as feature models, use case models, design models, component models and test models. Common parts are not part of the variability model at all.

A survey conducted in 2013 concluded that feature-based modeling of variability is most popular among practitioners [Ber+13]. An attempt is made to summarize the concepts as briefly and concisely as possible. The presentation makes no claim to be complete. For further information, please refer to the respective literature.

2.4.1 FODA

Feature modeling was proposed as part of the Feature-Oriented Domain Analysis (FODA) method by Kang et al. [Kan+90]. In FODA, a domain is defined as a set of current and future systems that share common capabilities. The goal of the domain analysis is to discover and represent commonalities and variabilities between them.

In [Kan+90, p. 3], the term *feature* is introduced as "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system". The resulting model is usually displayed as a feature tree with mandatory and optional features, as well as cross-tree dependency relationships (e.g. Feature A *requires* Feature B, or Feature A *excludes* Feature C). Mandatory features describe the common parts of a system, whereas optional features indicate its variation points. Later the representational possibilities were further extended, e.g. to include cardinalities by Czarnecki et al. [CHE05]. It is a representative for the integrated documentation of variability, where variability is modeled from the end user (stakeholder) perspective.

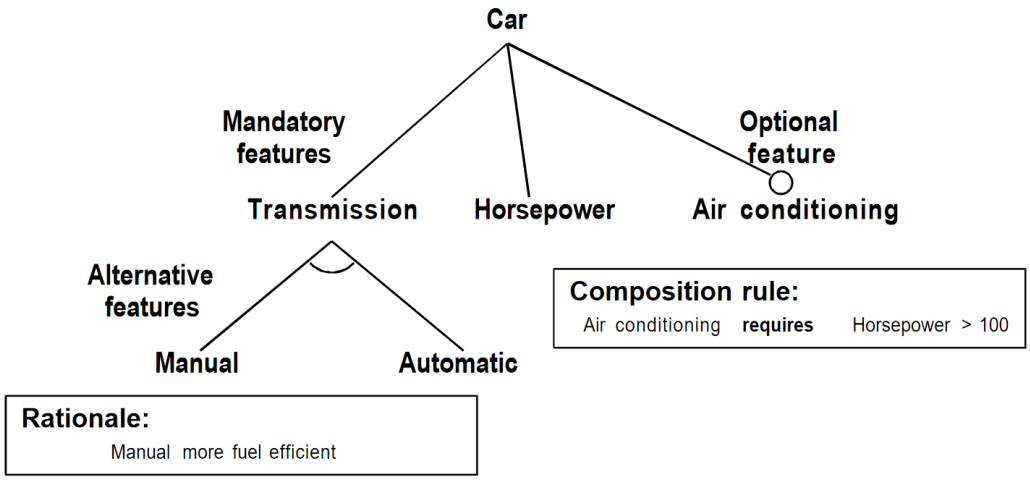


Figure 2.5: Representation of a feature tree according to FODA [Kan+90, p. 36]

Figure 2.5 shows an example of a simple feature tree. A car is shown with two mandatory features (Transmission, Horsepower) and one optional feature (Air conditioning). There are two alternatives for the choice of transmission, of which Manual is more fuel efficient. As an additional constraint, the selection of the AC requires the Horsepower to be greater than 100.

The representation is very clear and easy to understand even for non-professionals. The range of features, however, reveals nothing about the internal system implementation. Therefore restrictions are not always comprehensible: Why does the air conditioning system need more than 100 horsepower? The reasons for these restrictions remain hidden in the architectural implementation of the system and is deliberately excluded in this user-centric view.

2.4.2 OVM

The Orthogonal Variant Model (OVM) as described by Pohl et al. in [PBL05] captures all variability aspects (of software) in a separate model. Although the approach has its origins in software engineering, it can nevertheless be applied to systems engineering. OVM also uses feature trees to depict variability using variants, variation points, and variability dependencies among each other. But unlike FODA, the focus is on the variability aspect alone. The resulting orthogonal variability model is thus devoted primarily to capturing and representing variability and does not include any design decisions or feature descriptions. Pohl et al. state that "software development models (e.g. feature models) are already complex, and they get overloaded by adding the variability information" [PBL05, p. 75].

The orthogonal variability model however relates the variability defined by it to other (software) development models such as feature models, use case models, design models, component models, and test models [PBL05]. This is achieved by references e.g. *trace*, *implement* between the models. However, the consistency and correctness of these relationships poses a major challenge. Especially when the overall system is growing and several views emerge, each of which has references to the variability

model [Tru+10][Bil+18]. A practical application and extension of OVM was proposed by Dumitrescu. His CO-OVM (Constraint Oriented Orthogonal Variability Model) suggests a variant management metamodel for SysML models that allows an explicit representation of variability during system conception by focusing on requirements and constrains [Dum14].

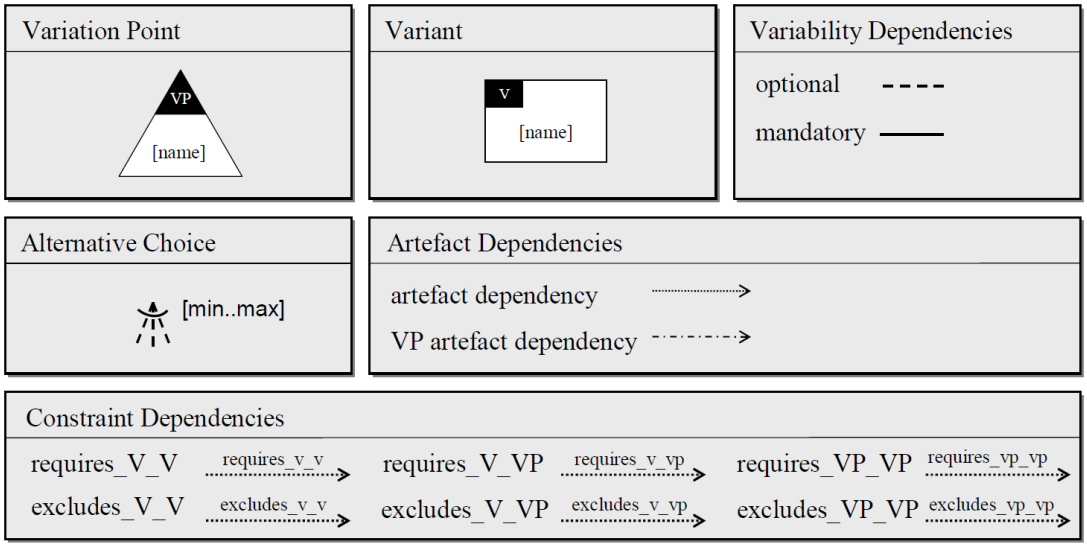


Figure 2.6: OVM notation [PBL05, p. 85]

Figure 2.6 summarizes the elements and notation capabilities of OVM. The vast majority of the identified case studies used OVM for variability modeling. One reason for this could be that it provides an independent description of variability and can also be applied retrospectively to an already existing system model. In addition, there is plenty of tool support that encourages the use of OVM notations.

2.4.3 CVL

The Common Variability Language (CVL) "is a modeling language to specify the variability aspect of any model that is defined based on the Meta Object Facility (MOF)" [Wei16, p. 44]. For this purpose, a base model e.g. created with SysML or any other Domain Specific Language (DSL) is extended with a variability model in CVL, which describes all variable aspects. An additional resolution model is then used to describe how the variable elements shall be resolved respectively realized. "A model-to-model transformation creates the resolved model in the same language as the base model, for example SysML" [Wei16, p. 44].

The concept and language of CVL was planned to be an adopted standard of the OMG. "Unfortunately the adoption process has stopped and it seems that CVL will not become a standard anymore" [Wei16, p. 44]. Although it has been mentioned in some studies and examined in more detail (cf. [Sve+10]), it will not be considered further in this thesis due to its declining attention and importance with regard to SysML.

2.4.4 VAMOS

Variant Modeling with SysML (VAMOS) is most recently described in Weilkiens book with the same name [Wei16]. The concept itself is not completely new, but was tailored by Weilkiens to SysML-specific needs. "It uses the profile mechanism of SysML to extend the language with a concept for variant modeling" [Wei16, p. 7]. The introduced stereotypes are described later on. A special tool support is not necessary and not intended, since the method aims to be completely SysML-compliant. Weilkiens argues that while the use of specific tools for variant modeling has its advantages (especially when the requirements for variant modeling are demanding), the price is paid with much higher effort and complexity. This is reflected in particular in the acquisition and maintenance costs of the tools, as well as their training.

In essence, the VAMOS approach consists of the following three different types of elements. The representation of all components and their relationships is usually done in package diagrams.

The Core. The core contains the "normal" system model, which is independent of the variant aspects except for the assignment of variation points. Variation points are elements (docking points) of the core, which are refined by variant elements. The core therefore does not specify a particular system, but can be seen as a toolbox that allows many different types of systems depending on the selection (configuration) of these variant elements [Wei16, pp. 11-20]. Variation points can therefore be seen as interfaces to the core, allowing it to be customized.

The Variants. "A variant is a complete set of variant elements that varies the system according to a variation. A variant is also known as a feature of the system" [Wei16, p. 4]. Multiple manifestations of a variant are summarized under one variation. These relationships can be represented as a feature model/tree analogous to the OVM and FODA approaches mentioned above. The variations form the branches and the variants form the leaves of the feature tree. Additional properties of the variation packages allow the modeling of constraints and cardinalities. Between variants there can also be XOR or REQUIRES dependencies [Wei16, pp. 22-25].

The Configurations. "A variant configuration is a valid set of variants and the core" [Wei16, p. 5]. A configuration can be conveniently displayed as a matrix view, that shows the selection of a specific variant for each variation point. It therefore binds each variation point of the core to a specific variant and thus ultimately defines a specific system variant.

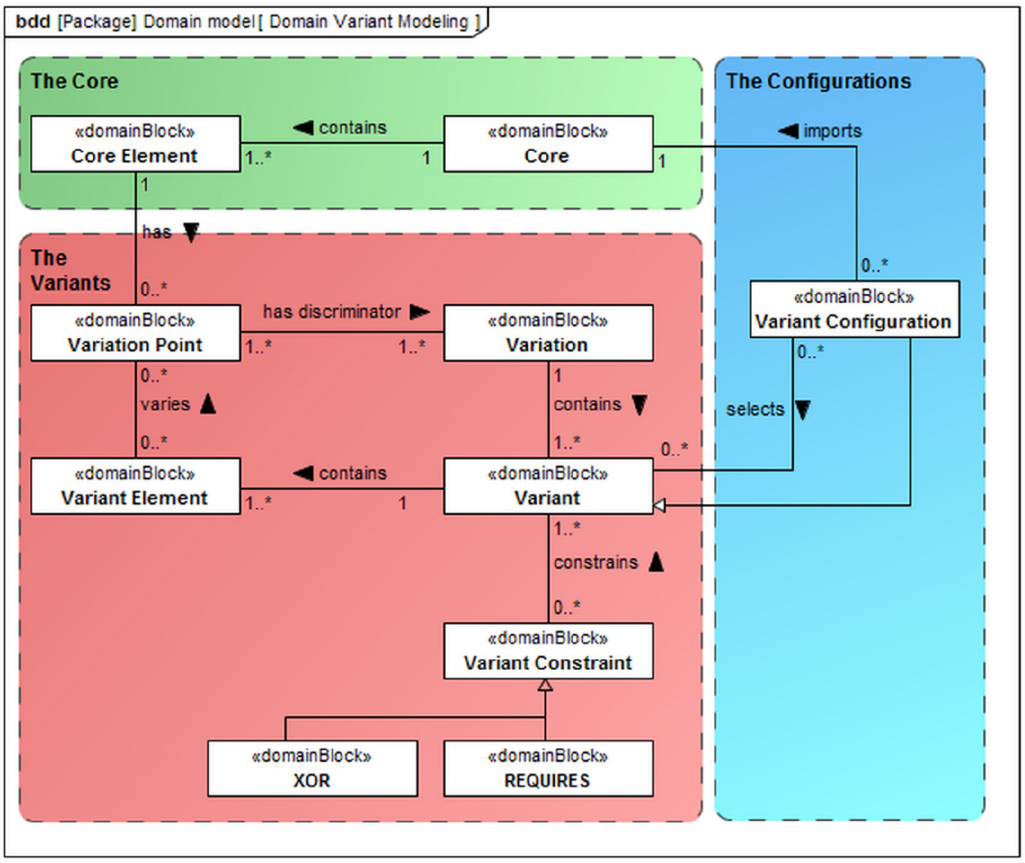


Figure 2.7: Weilkiens conceptual model of VAMOS [Wei16, p. 4]

Figure 2.7 shows the conceptual model of the elements used in VAMOS and their relationship to one another. The three parts previously outlined are shown in blocks of different colors. The picture illustrates how the configuration binds the core and its variations together. A variation is the discriminator for variants. The structure is recursive, so a variant can again include variations. In its basic concept it is similar to CVL - both separating the common part from the system parts that are only valid for a specific variant. The main difference is that CVL stores the core part and the variant part in different models. In VAMOS both parts are in the same model but separated by their package organization.

Although VAMOS can be used to realize variability, it was deliberately included in this section. This is because of its strong structural and packet-oriented nature. It does not provide support for modeling behavioral variability. Considering this limitation to structural variability, however, VAMOS could still be regarded as a realization mechanism with a low granularity or a high level of abstraction. Other research, though, shares the classification as a method that primarily serves the visualization of variability (cf. [Bil+19]).

Nevertheless, this approach can be sufficient and practical, especially for smaller systems, because it does not require any other tool and allows the direct integration of variability information into the SysML model. This corresponds to the approach of integrated documentation according to Metzger and Pohl [MP14].

2.5 VARIABILITY REALIZATION

This section focuses on approaches and concepts that deal with variability in the solution space. Both, approaches from the literature and those that have already been successfully applied in practice are presented. The focus is on the question of how variability can be implemented at the element level.

For this purpose, a short overview of approaches from related engineering disciplines is given first. A review of approaches from the field of systems engineering concludes the section. The transfer of the approaches to the model level and their evaluation can be found in Section 3.4 as part of the conceptual model.

2.5.1 SOFTWARE ENGINEERING

Compared to its siblings, software engineering is a relatively young engineering discipline. However, early on in its development, there was already a strong focus on reuse of (code) artifacts. Design for and realization of reusability are part of numerous proposed and meanwhile well established concepts and methods.

The relatively low barrier of being able to incorporate far-reaching changes even late in the development process makes many of these mechanisms particularly interesting for software engineering. Even existing code can be refactored by these measures, which also leads to a widespread use. Although it is not advisable to consider reusability in retrospect, there is still almost always the possibility.

Component-based software engineering was proposed at a conference in 1968 by McIlroy to cope with the ever-increasing complexity of emerging systems [McI69]. By dividing software into components, their reusability and overall quality ought to be increased. Components must be clearly documented, well tested and thus be robust even against unintended use. This principle is still followed today. Frameworks facilitating this principle are for example the Component Object Model (COM), .NET or the Common Object Request Broker Architecture (CORBA).

Over the time many design patterns emerged, that provided reusable solutions for recurring problems. Although originally proposed as purely architectural concepts by Christopher Alexander, these design patterns gained more popularity in computer science after the book *Design Patterns: Elements of Reusable Object-Oriented Software* was published by the so-called "Gang of Four" in 1995 [Gam+95]. The consistent and widespread adoption of these patterns has further sharpened the view of an entire community on common and interchangeable interfaces/classes. Stronger attention towards object-oriented programming also contributed to this. Later guidelines, including the very famous *Clean Code: A Handbook of Agile Software Craftsmanship* [Mar08], still promote many of these patterns to improve the readability and reuse of newly developed as well as refactored code artifacts.

Software Product Lines (SPLs) sometimes also referred to as *Software Families* have an even stronger focus on reusability. By separating application and domain engineering, this method provides tools and techniques for creating a collection of similar software systems from a shared set of assets using a common means of production (cf. SEI[11]).

11 Software Engineering Institute - https://www.sei.cmu.edu/

Key elements are the separation of common and product-specific components as well as the clear identification of variability within the product line. Especially in the predictive planning of software reuse, SPLs differ from previous approaches, which were rather opportunistic in this respect. From these basic ideas of SPLs, PLE emerged as a generalized, not just software-specific method for managing reuse and variability.

According to Trujillo et al. SPL approaches can be broadly categorized in two main groups depending on how they express variability in software artifacts. In *compositional approaches*, also known as positive variability, the variable parts are encapsulated in modular units which are put together according to the features selected for building a system. In *integrative approaches*, also known as negative variability, the artifacts contain both the common and variable parts. Building a system means keeping the variable parts of the desired features in the artifacts while removing those parts belonging to unselected features [Tru+10]. This corresponds to the terms *Substitution* and *Selection* as introduced in Section 2.2.

The following list contains the most frequently mentioned realization mechanisms for variability in the context of software engineering [PBL05][Pat11][Ape+13][ZDB16]:

- Cloning
- Conditional Compilation
- Conditional Execution
- Polymorphism
- Module Replacement
- Aspect-Orientation
- Frame Technology

Apel et al. differentiate between *Annotation-based* and *Composition-based* approaches. Annotation-based approaches annotate a common code base, such that code that belongs to a certain feature is marked accordingly. During product derivation, all code that belongs to deselected features or invalid feature combinations is removed (at compile time) or ignored (at run time) to form the final product. Examples of these are Conditional Compilation or Conditional Execution. In practice, annotation-based approaches are widely used due to their simplicity and the fact that they are already natively supported by many programming environments and languages. Composition-based approaches implement features in the form of composable units, ideally one unit per feature. During product derivation, all units of all selected features and valid feature combinations are composed to form the final product. An example is a framework that can be extended with plugins. In principle, any combination of both approaches is possible as well [Ape+13, p. 51].

Patzke introduced tactics that are helpful in classifying and characterizing the mechanisms as well as pointing out important aspects to consider when using:

Increase VP explicitness. Increasing the visibility of variation points (VPs) makes variants easier to detect and product line assets easier to evolve.

Allow appropriate variant granularities. Associated variant elements of different sizes should be realized by mechanisms that support a corresponding spectrum.

Limit late binding. Later binding times lead to less degree of freedom for realizing variants.

Isolate variants. Separating common and variant modules allows them to evolve independently.

Provide automated production. Automation reduces application engineering effort.

Provide defaults. Defaults reduce the number of variants.

This list of criteria was extended by Zhang et al. in 2016 and resulted in the characterization of the mechanisms shown in Figure 2.8.

Variability Mechanism	Techniques	Binding Time	Granularity	Explicit Variation Points	Variant Isolation	Open Variation	Non-Code Artifacts	Defaults
Cloning	Copy, branching	Construction time	Any	Implicit	Yes	No	Yes	No
Conditional Compilation	Preprocessor	Construction time	Any	Explicit	No	No	Yes	Yes
Conditional Execution	Cond. statements	Run-time	Limited	Implicit	No	No	No	No
Polymorphism	Function pointers, overloading, etc.	Mostly run-time	Limited	Implicit	Yes	Yes	No	No
Module Replacement	Build system	Mostly constr. time	Limited	Implicit	Yes	Yes	Yes	No
Aspect Orientation	Code weaving	Mostly constr. time	Limited	Implicit	Yes	Yes	Yes	Yes
Frame Technology	Frame adaptation	Construction time	Any	Explicit	Yes	Yes	Yes	Yes

Figure 2.8: Variability Mechanism Characterization [ZDB16]

Technique. Links the abstracted variability mechanisms to their actual implementation, i.e. their respective development or programming technique. Some techniques are supported by almost all programming languages, others can only be applied in specific languages [ZDB16].

Binding Time. Describes the time at which the selection of a variant occurs. While most mechanisms define their VPs and variants at the time of creation/construction, some allow dynamic selection at runtime. Early binding time resolve the variability configuration space early and potentially optimize running efficiency, while mechanisms with late binding time provide more flexibility [ZDB16].

Granularity. Defines the level up to which elements/components can contain variation points. While text-based mechanisms can basically support any granularity of code variants, other mechanisms enforce a certain size or form of the variants within the code structure (e.g., a function, a class, or a file) [ZDB16].

Explicit VPs. Describes whether a variation point is clearly recognizable as such. Since variability is a cross-cutting concern with (often highly) scattered realization code, mechanisms with implicit variation points tend to cause challenges in development and maintenance [ZDB16].

Variant Isolation. While Patzke mentions the isolation of variants and describes them also as open or closed, Zhang et al. extend this aspect and consider both aspects separately. Isolation refers to the categories of annotation-based versus composition-based approaches, e.g. whether all variants are structured in separate modules or combined in a single file. Code using annotative mechanisms is more integrated but potentially more complex, while the variability code using compositional mechanisms is less complex but more fragmented [ZDB16].

Open Variation. Reflects the extensibility of a variation point, i.e. whether the VP can be extended independently by attaching external modules (plugins, etc.). In case the variable code must be compiled together with the core code, extending is not possible.

Non-Code Artifacts. Indicates whether this mechanism can be used for other artifacts besides code, e.g. for data files, models or text files.

Defaults. Ability to specify a standard selection. If no selection is specified for a VP, the standard variant is chosen. This simplifies the variation logic and reducing the number of variants (by one) [ZDB16].

2.5.2 ELECTRICAL/MECHANICAL ENGINEERING

While many mechanisms for realizing variability can be found in the literature that focus on software engineering, it should be mentioned that of course other engineering disciplines, such as electrical or mechanical engineering, have also developed methods to deal with the increasing number of variants.

In electrical engineering, there is a trend towards modular hardware design. Lower component prices and the large number of commercial off-the-shelf (COTS) components are driving this development. The modular ideology divides a system into several independent parts, each of which can also be used in other systems. Open-source hardware platforms are gaining momentum, particularly in single board computer and development board design. As far as variability of these products is concerned, the focus is on well-defined and consistent interfaces. Components can be exchanged, replaced or extended, possibly even manufacturer-independently. Also, there is still a recognizable trend towards gradual merging of hardware and software. The use of Field Programmable Gate Arrays (FPGAs) delays the decision which functions are ultimately to be performed by hardware or software. Hardware description languages such as VHDL[12] or Verilog allow the realization of functions in hardware at a very late stage (cf. [Lie+09]). At component level, many manufacturers offer microcontrollers of various configurations in pin-compatible packages. Within a product family, for example, the memory can be easily increased even after the PCB has been layouted.

In mechanical engineering, design reuse is also a subject of research. Compared to software engineering, however, the reuse of mechanical (or electrical) components is much more complicated, since they are physical components that are more difficult to modify and customize. Often large quantities are manufactured or ordered and stored to reduce costs. While processors can be reprogrammed, the rework of other stored components causes high costs or is not possible at all. As these retrofits are costly and often difficult to implement, this means that "compatibility among interacting components requires more consideration" [ONX08, p. 18]. Approaches to achieve reuse include the use of standard components. "Modular design is [also] an established method in this domain" [ONX08, p. 18]. Similar to electrical engineering, the emphasis here is on standardized interfaces (e.g. mounting points, adapter plates).

In the case of computer-aided design, however, most of the methods and approaches from software engineering apply. 3D models and layouts can be reused, copied and

12 Very High Speed Integrated Circuit Hardware Description Language

extended, as they represent digital artifacts (similar to code). In mechanical engineering, or to be more specific for mechatronic systems, a solution pattern based on MBSE was proposed by Anacker et al. [Ana+13][ADG14]. This methodology identifies solution knowledge and process solution patterns from existing mechatronic systems to provide meaningful reuse during the design of new systems. Through this process, as much (even implicit) knowledge as possible should be preserved.

2.5.3 SYSTEMS ENGINEERING

The literature review has identified some approaches in the field of systems engineering. However, the majority of publications deals with the modeling and representation of variability instead of its implementation at the model level. The following is a brief summary of publications that address the implementation to some extent. In particular, the practical application and the difficulties identified can be helpful in the later evaluation of the different approaches.

In an exploratory case study, Berger et al. present industrial practices of three companies that use variability modeling and thus show how modeling is applied in industry. According to them, there is a large gap between the theoretical methods and their practical application in everyday use. The primary benefit of variability modeling lies in variability management – organizing, visualizing, and scoping features – less in configuration and automation [Ber+14]. All three cases mainly use Conditional Compilation as a mechanism to realize code variability.

A recent study from Bilic et al. describes a tool-supported approach that allows to annotate SysML models with variability data. This variability information can then be exchanged between a system modeling tool and a variability management tool using the Variability Exchange Language (VEL) [Bil+19]. By using the proposed tool chain, variant realization (with SysML) and variant management (in pure::variants (p::v)[13]) are seamlessly coupled. The study was demonstrated through an example from Volvo CE[14] and suggests a potential improvement in efficiency.

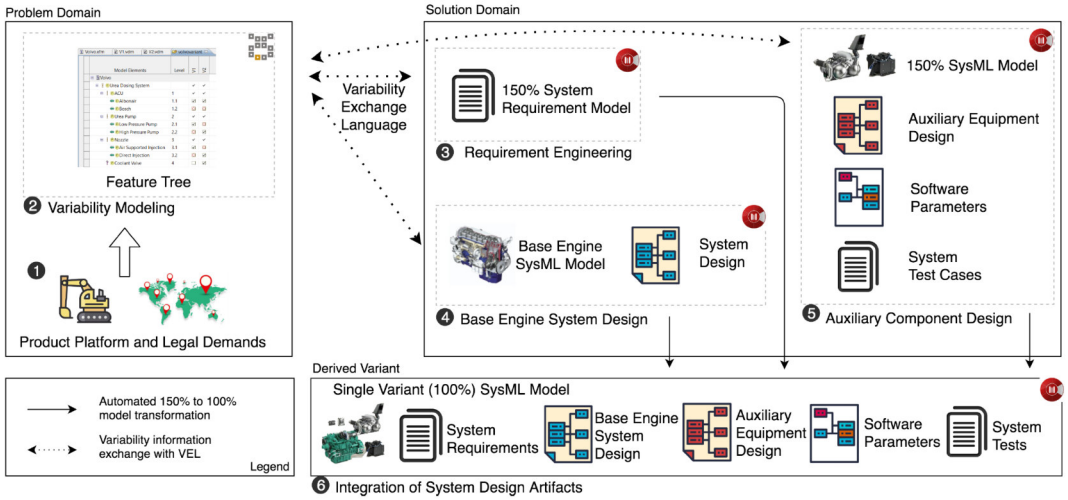


Figure 2.9: Model-based Tool Chain [Bil+19]

13 pure-systems GmbH - https://www.pure-systems.com/
14 Volvo Construction Equipment - https://www.volvoce.com/

A schematic representation of the process is depicted in Figure 2.9. The variability in the Problem Domain (Variability Management) is handled with a dedicated feature model (in pure::variants). In the Solution Domain (Variability Realization) a single, over-exposed (150%) model is used. VEL is used to exchange variability information between these two representation and allows automatic traceability between feature model and implementation. As a result and with an extension of the modeling tool used (Modelio[15]), a specific 100% model can be automatically generated for any particular variant. This study is particularly interesting because it focuses on SysML and uses a very similar approach to that in the feasibility study (cf. Chapter 4).

A study conducted by Tolvanen and Kelly investigated DSLs and especially how they address variability in product line development [TK19]. For this purpose 23 different cases were examined. Various company sizes, languages and models from a wide range of fields were examined. In particular how they apply reuse and production line approaches. About 50% of the identified approaches are currently not (yet) used in real projects, because they are still in an evaluation and testing phase. Nevertheless, especially the diversity of these approaches shows how difficult a generally valid solution is and how different the requirements are. Since the study focuses on domain-specific modeling languages, SysML is not part of it.

The fact that some studies have only recently been published and were supported by notable companies shows that, on the one hand, current research is addressing these issues and, on the other hand, that there is a growing need to find practicable solutions on how to implement variability.

15 Open source extensible modeling environment - https://www.modelio.org/

3

CONCEPTUAL MODEL

This chapter describes the conceptual model and thus all aspects that were further investigated in the context of this thesis. The topic is approached from the outside to the inside. For this purpose, the causes for variants are first examined and described in detail. Afterwards, the approaches for variation management already introduced in Section 2.1.2 will be further elaborated. The selection of such an approach already has far-reaching effects on the possibilities for variability realization. Subsequently, the possibilities and limitations of tool-supported variant modeling are discussed. The main part includes an overview of existing mechanisms for the realization of variability with special emphasis on the modeling aspect. A presentation of KPIs, with which mechanisms can be evaluated, compared and thus be chosen, concludes the chapter.

3.1 VARIANT DRIVER

Two aspects significantly influence the creation of variants. In the following, these drivers are discussed in more detail. While variability describes the intentional co-existence of different variants at the same point in time, evolution i.e. the implementation of changes over time, leads to new variants (revisions) as well.

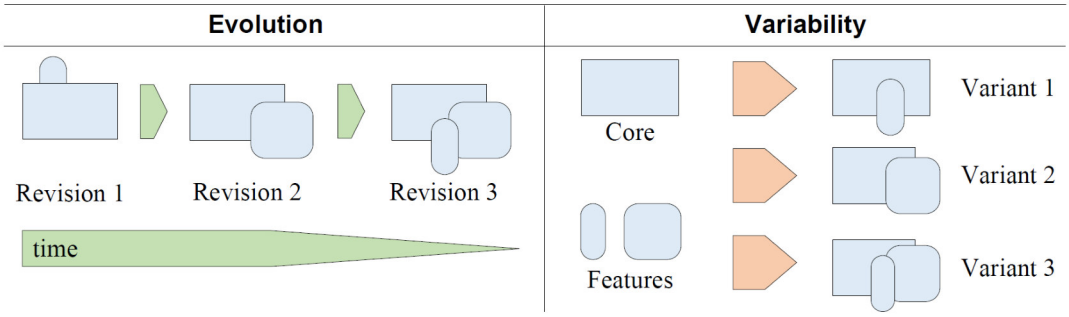


Figure 3.1: Evolution and Variability [Sch18, p. 6]

Figure 3.1 shows both of these driving factors. While product/system evolution over time is principally inevitable, variability is somewhat controllable. However, there are circumstances that cannot be foreseen or controlled.

Evolution leads to variation in time, variability leads to variation in space. At a first glance, they seem orthogonal. But, since each new variant is once again subject to temporal variation and the evolution of a domain assets can have in turn an impact on all variants, there are situations where they overlap. In practice, both aspects must be considered equally.

3.1.1 VARIABILITY

Variation points are required wherever a system feature may have different characteristics. Within the context of Product Line Engineering (PLE), these variation points can be used to create new variants. Usually, this is the result of continuous product improvements and portfolio expansions. Meaning that a product is to receive a new feature or an existing feature is changed. The reasons for this can be of different nature. But what these types of new variants have in common is their deliberate creation and a changed configuration of internal properties. Examples of this are:

- additional new feature (technological innovation)
- new variant with a subset of functionality for a different price segment

In contrast, there are also often drivers for variability that do not occur intentionally or cannot be planned. Thus, it is especially important to become aware of these drivers in system modeling, e.g.:

- new regulations (laws, standards)
- changed environmental constraints
- change in a used third-party component (interface, API, discontinuation)

Even if the triggers themselves are not predictable, it helps to become aware of their existence. Especially those that can potentially lead to *frequent* changes.

The variant drivers listed below are characterized by the trigger that requires a system change and thus a new/changed variation point. The examples listed here are intended to provide a comprehensive overview of the different areas where changes occur and new variation points can arise. There is no claim to completeness.

SYSTEM CONTEXT. Many changes can occur in the system context. New system components (participants/actors/peer systems) can be added or modified. For this purpose, existing interfaces may have to be extended or new ones added. Unintentional changes can also occur in interfaces to external systems, since these are not exclusively controlled by the system. This type of externally influenced system adaptation often occurs in the IT environment. Newly discovered security vulnerabilities or changed Application Programming Interfaces (APIs) to third-party systems force developers to modify their systems accordingly to remain functional. If the system is not designed for such changes in the system context, this can have far-reaching consequences for its durability and usability. Changes in environmental conditions also fall into this category because they define the system boundaries. Such changes can also have various effects on the overall system and its implementation.

EFFECT CHAIN. Effect chains include both functions as well as their relations. A change in the effect chain creates new functionalities simply by rearranging or combining existing signals. No new components are involved. For example, input signals can pass through different processing steps depending on the product variant. The implementation of a second algorithm would also be an example. Algorithm A is less accurate, but requires less performance. Algorithm B is more accurate and requires a

more powerful processor. Depending on the product variant and its configuration, the appropriate algorithm must be selected. Depending on this selection, some elements, interfaces and connections remain unused.

INFORMATION FLOW. A change in the information flow affects first of all the interfaces of the system or its components. These changes can subsequently lead to other changes, such as alternative processing paths (see Effect Chain) or changed behavior (see Behavior). Information flow, however, primarily concerns a change in content, such as a changed data type.

BEHAVIOR. Trigger for the system change is the need to behave differently. Behavioral change is mainly reflected in the behavior diagrams (Activity, Use Case, Sequence and State Machine). Therein they lead to additional states, transitions or messages. The correlation between intended and modeled behavior is very clear and straightforward. Nevertheless, a change in behavior may also affect other aspects of the model or, vice-versa, might haven been triggered by them.

ELEMENT PROPERTIES. A system change is triggered by a variation of its properties. These can be system-level properties or requirements (performance, reliability, etc.) or the properties of individual system components. A change refers both to the attributes themselves and to their assigned values. For example, a temperature sensor could have a parameterizable threshold for triggering an interrupt or not. This would make the attribute itself optional. If the threshold is now varied, an additional variation point at the parameter level is created. With the help of parametric and requirements diagrams, these configurations and variations can be well documented, but they often have far-reaching consequences for other model components as well.

ALLOCATION. A change in allocation occurs when tasks and responsibilities are reassigned. For example, functions that were previously performed by one or more components are rearranged and now performed by another component. As a result, relations between system components are also subject to change. Moving a complex calculation to another processor with less workload would be an example. The increasing integration of multiple functions into one chip (SOC[1]) is another example of how allocation varies. In this case several components are replaced by one. This is often the result of cost reduction or technical innovations.

COMPOSITION. This refers to changes to the system structure and the assignment of responsibilities. Decisions to remove optional components or extract commonalities in libraries are examples. These types of changes have far-reaching effects on the decomposition of the entire system.

BUG FIX / MAINTENANCE. Identified defects and/or improvements to already existing functionalities can always be a reason for changes. The affected model areas can vary as well as the implications for them. Initially, it must be identified which variants are affected in order to be able to implement a system-wide or variant-specific correction. The changes caused by defects do not usually lead to a planned new

1 System on a chip

product variant, but merely correct functional deficits of an already existing variant. Thus, bug fixes are not to be understood as a new feature but are rather a variant evolution.

3.1.2 EVOLUTION

As mentioned before, change is inevitable. Developers are humans and therefore make mistakes. Furthermore, it is almost impossible to foresee changing internal or external requirements, or new functionalities demanded by users, at the time of deployment [Sch18].

There are several approaches to classify product line evolution scenarios. A straightforward and easy to understand classification scheme was introduced in [Zur14, p. 62]. It differentiates the scenarios according to the trigger or "drivers of change".

(1) Maintenance
   a) Bug Fix
   b) Change (requirement, environmental constraint, etc.)

(2) Evolution
   a) add new feature
   b) remove existing feature
   c) create new variant (select, add and change features)
   d) remove existing variant (and associated features)

While *Maintenance* in this classification can be regarded similar to variability, *Evolution* focuses on the further development of the variation points within a product line. As soon as a product line evolves, the interaction of its common and variable parts usually has to change as well. The adaptation is done by a family engineer using an appropriate variability mechanisms. It is necessary to apply these mechanisms in a disciplined manner to minimize further complexity. A change in common parts may change other variants as well. The decision to implement a change in a common part or only specifically for one variant has a major impact on the product line. Apart from these two possibilities, there are alternatives, e.g. implementing the solution as another domain asset and selecting that one for this variant by means of a new variation point. This allows to decide for each variant which implementation to use. The structure of the product line has a great influence on the possibilities. In case of a production line (150%), application engineering is omitted and any change has to be integrated with a new variation point if not intended for all variants.

A set of basic evolutionary scenarios covering the most important types of changes to a product line has been developed by Patzke [Pat11]. The remedy proposed in his work is to capture evolutionary steps and reapply them later when a similar developmental situation occurs. Especially the ability to transfer the described evolution steps to elementary realization activities are of interest, since this could also be applied to SysML elements as well. The feature evolution described by Patzke is shown in Figure 3.2. It summarizes the most atomic evolution possibilities that a product line asset can

undergo. Figure 3.2a depicts the changes to the features, while Figure 3.2b shows the changes to the corresponding (code) artifacts. A list of the captured product line evolution scenarios follows in Table 3.1. Although Patzke created this list for code artifacts, all aspects can nevertheless be applied to the model level.

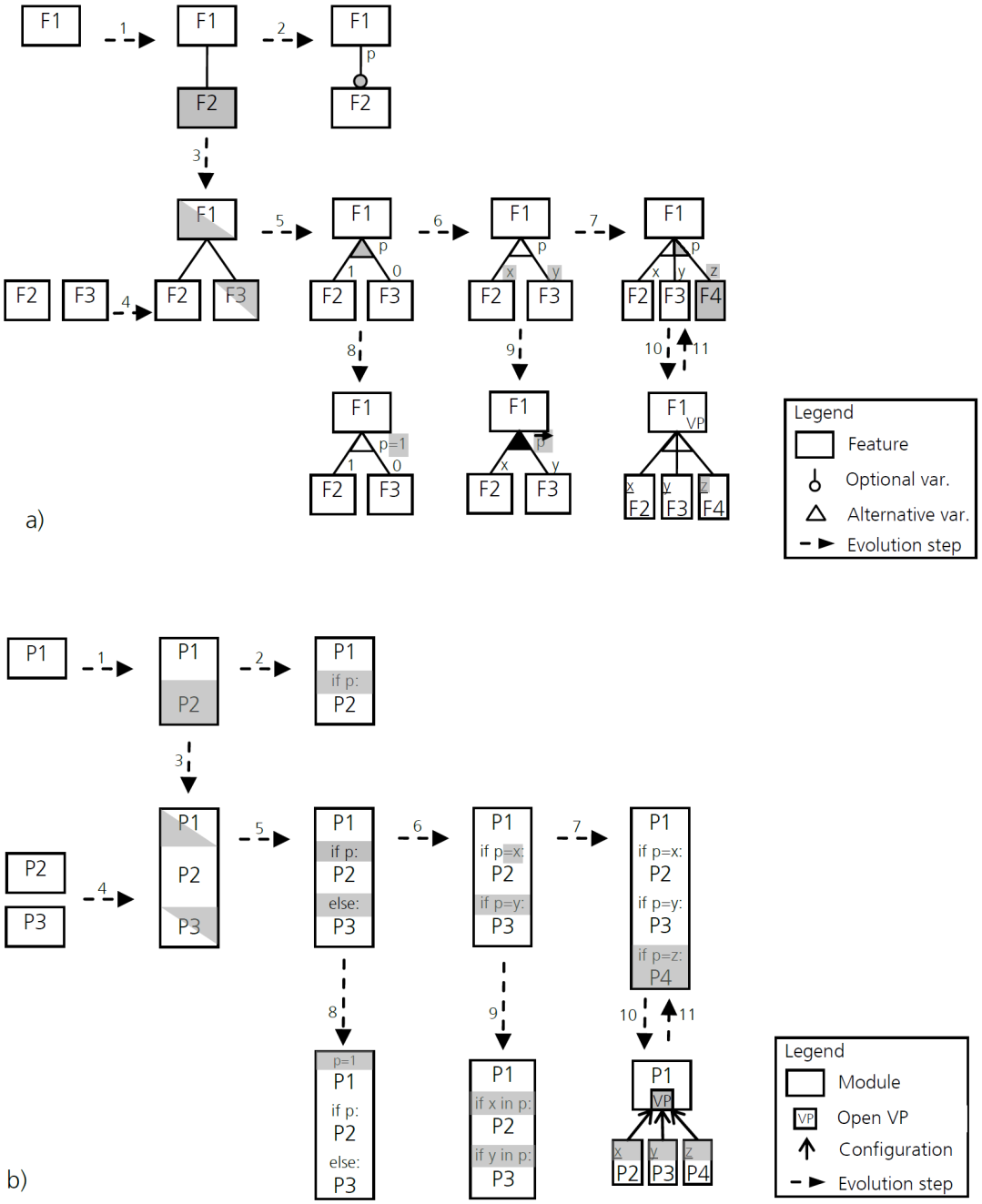


Figure 3.2: a) Elementary feature evolution b) corresponding pseudocode [Pat11, p. 135]

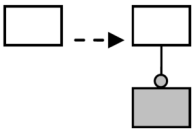
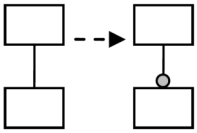
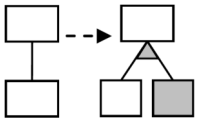
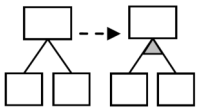
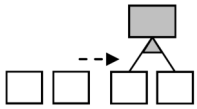
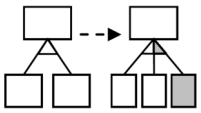
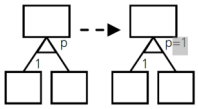
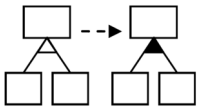
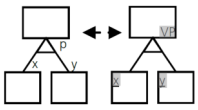
ID	NAME	FEATURE EVOLUTION	STEPS (CF. 3.2)	SCENARIO
ES1	optional feature creation		1, 2	realize a new feature which depends on an existing feature
ES2	optional variation point creation		2	make an existing feature an optional variant element
ES3	alternative feature creation		3, 5	create an alternative feature as a substitute for an existing common feature element
ES4	alternative variation point creation		5	make existing alternatives more explicit by realizing their common variation point
ES5	common feature extraction		4, 5	consolidate common elements and converting them into alternatives
ES6	alternative feature addition		(6), 7	realize an additional alternative feature by extending an existing variation point
ES7	default addition		8	reducing complexity by adding a default selection
ES8	addition of multiple coexisting possibilities		9	make several features, that have been alternatives previously, available simultaneously
ES9	variable feature extraction / inlining		10/11	both methods concern the coupling of variants; depending on the total number of variants, it may be necessary to convert them from closed to open variation or vice versa

Table 3.1: Product line evolution scenarios captured from Figure 3.2 [Pat11, pp. 137-143]

ES1 *Optional Feature Creation* implements a new feature, e.g. a new functionality. Afterwards, products containing the pre-existing features or extended by the new (optional) feature should be producible. The simplest approach is to clone an existing element and modify it to realize the new feature. A less intrusive method of implementation would be Conditional Compilation.

ES2 *Optional Variation Point Creation* differs from *Optional Feature Creation* as the new feature already existed as an element of the common code but was not explicitly designed as a variation point. Language-agnostic variability mechanisms, such as Conditional Compilation, tend to be the best choice for realizing this scenario.

ES3 *Alternative Feature Creation* substitutes an existing common feature with an alternative. This may be necessary, for example, if a new feature is required but existing elements cannot (yet) be removed or replaced. The simplest approach is a two-step process. First, the common element for which an alternative is to be provided is identified. Second, this element is made a variant by introducing an alternative variation point. The creation of variants can be achieved with Cloning or Conditional Compilation. For variant selection Conditional Compilation is most appropriate.

ES4 *Alternative Variation Point Creation* creates new alternatives from already existing elements, which, however, did not previously represent variability, by recognizing and explicitly realizing their common variation point.

ES5 *Common Feature Extraction* is a consolidating scenario to eliminate duplicate elements. First, common elements are extracted and differing elements are then converted into alternative.

ES6 *Alternative Feature Addition* is the extension of an existing variation point by a new alternative.

ES7 *Default Addition* is an optimization scenario that favors one alternative over others by declaring it a default. This ultimately reduces complexity, since no selection has to be made for this variation point. The default selection can be overwritten, but if this is not done explicitly, the default selection will be chosen.

ES8 *Addition of Multiple Coexisting Possibilities* becomes necessary when several features that were previously alternatives must be available simultaneously. The selection of alternatives is no longer limited to an exclusive disjunction.

ES9 *Variable Feature Extraction / Variable Feature Inlining* are two (complementary) scenarios which are usually applied to alternative or coexisting features and concern their coupling resp. isolation. Depending on the number of variants, it may make sense to keep them together in one module or develop them further separately.

[Pat11, pp. 138-143]

3.2 VARIATION MANAGEMENT APPROACHES

This section deals with the question of how system-wide variability and reusability of components can be implemented. Methods for implementing reusability in the PLE context were already introduced in Section 2.1.2. At this point, three of these concepts will be discussed in more detail and the resulting possibilities for variant realization will be investigated. Since the methods are rather abstract and coarse-grained, it could be assumed that they have little or no influence on concrete variation points. However, restrictions and general stipulations can already result from decisions made at this level, limiting the choice of tools and mechanisms for later implementation.

It has already been shown which causes can lead to a large number of variants. If a high degree of flexibility is required for a system or if many changes are foreseeable, an approach should be chosen as early as possible to deal with the resulting number of variants [Lab17]. While it seems difficult to decide on a method in advance and to apply it consistently, there are some aspects that can be helpful in the selection process even at an early stage. If necessary, deviation from the method can always be made at certain points. The following considerations are intended to assist in determining which approach may be suitable and when.

3.2.1 MANAGED CLONING

A new variant is created by copying an existing model or parts thereof, which are then modified to suit the specific requirements of the new system variant. Cloning can generally be applied to any artifacts and therefore appears again in the list of realization mechanisms. The term *Managed Cloning* is used to distinguish it from the ad-hoc *Clone & Own* approach and to emphasize the intended degree of structuring. In principle, the advantages, but also the limitations of cloning and managed cloning are alike. However, especially at system level, this approach should not be applied without careful consideration.

Systematic use. The use of managed cloning is no different from normal cloning. However, since the duplication of components and their independent evolution leads to a loss of relatedness, an important aspect of managed cloning is the maintenance of traceability and the visualization of changes. This is intended to ensure that relations and similarities between clones can still be identified in retrospect. In software engineering, this principle has already been used for a long time. Version Control Systems (VCSs) such as SubVersion[2] or Git[3] are used to store all changes to an (code) artifact into a repository. Each modification is assigned a unique revision number and contains information about the changes, the creation date and the author. Applied to a model, this allows concurrent accessibility for all users and a clear presentation of the history of changes. Derived clones are branches of the original model and can be modified independently, but still have information about their origin. Later on, entire branches, or only certain features of them, known as "cherry picking", can be merged back into the *main* model. This means, for example, that the current version of a model can be branched off and a change can be implemented for test purposes.

2 https://subversion.apache.org/
3 https://git-scm.com/

In case the result is satisfactory, all changes can be committed and merged into the main model. If the test fails, the branch has no immediate effect on the main model and can simply be discarded.

Benefits. Although not as easy as clone % own, using managed cloning is still relatively simple and intuitive. Because changes are stored in a repository (and can be reverted), there is very little risk involved. Correctness can be further increased by a review step. After a change has been committed, it must first be checked and released by a different person. Since the variants are developed in isolation, there is no risk of unintended side effects. The main advantage, however, is the possibility to work on the model distributed and in parallel. Everyone can access the most recent data and view/track changes made by others. A graphical visualization of differences is especially helpful in this case. If the change to a model can be considered as an atomic step that has been formally described and documented, then this step can be reapplied (to other models or components) and thus also be reused. This so-called patching can be helpful, for example, when recurring model changes have to be implemented to multiple variants. Strictly speaking, all these advantages have little to do with cloning itself - they are advantages that come from using a VCS. These advantages remain the same, regardless of whether the variants were created by cloning or in any other way. Nevertheless, the use of a VCS allows and encourages cloning of models at system level.

Limitations. The introduction of a VCS requires planning, training and consistent utilization. Training and the creation of a common understanding is often neglected. However, this is especially important in daily use. Apart from the technical requirements (server, administration, access rights, security), the greatest difficulty is the resolution of conflicts. The opportunity to make concurrent changes to the system model inevitably leads to these conflicts when merging. While graphical tools can help to solve the conflicts, it is nevertheless a difficult, time-consuming and error-prone task. It requires a mutual understanding to merge both your own and other people's modifications. The limitations therefore mainly result from the capabilities provided by the tools used. Since many VCSs come from the source code domain, they are specialized in processing text files. Models can be stored and managed as artifacts, but tools like *diff* and *merge* do not work with binary files. Special tools are therefore required. These tools must highlight the differences between the models, such as missing or changed components, values or dependencies. They must also support the user in making decisions during the merging process and ideally be fully integrated into the modeling tool. It is important to point out that while tool support can help to keep track of the clones, it does not indicate how strong the similarities remain. Suppose a defect is discovered in a component that has been copied multiple times. By using a VCS, the variants that contain this component can possibly be identified. However, the bug fix must be manually applied to all variants and (much more importantly) separately adapted and validated. It should also be noted that the consistent management of clones is necessary right from the product/project start. A later use is not at all, or only very difficult, to realize. A study by Rubin et al. shows the (re-)creation of a (software) product line from variants that have been realized by cloning. These variants were transferred either into libraries or into a 150% model in order to be able to manage them. A process that can take years depending on the number of variants [RCC13].

Example. LemonTree[4] is a proprietary tool that offers a large range of functionality for the implementation of managed cloning. It combines proven techniques and features of a VCS with a full integration into Enterprise Architect (EA). This enables collaborative modeling and distributed work while keeping track of all changes. Most importantly the diff tool can visualize conflicts and differences graphically (cf. Figure 3.3). All impacted elements are listed and a merge preview allows to examine the resulting diagram immediately. Conflicts that arise can thus be solved with a simple selection mechanism ("take theirs" vs. "take mine"). It supports different types of diagrams.

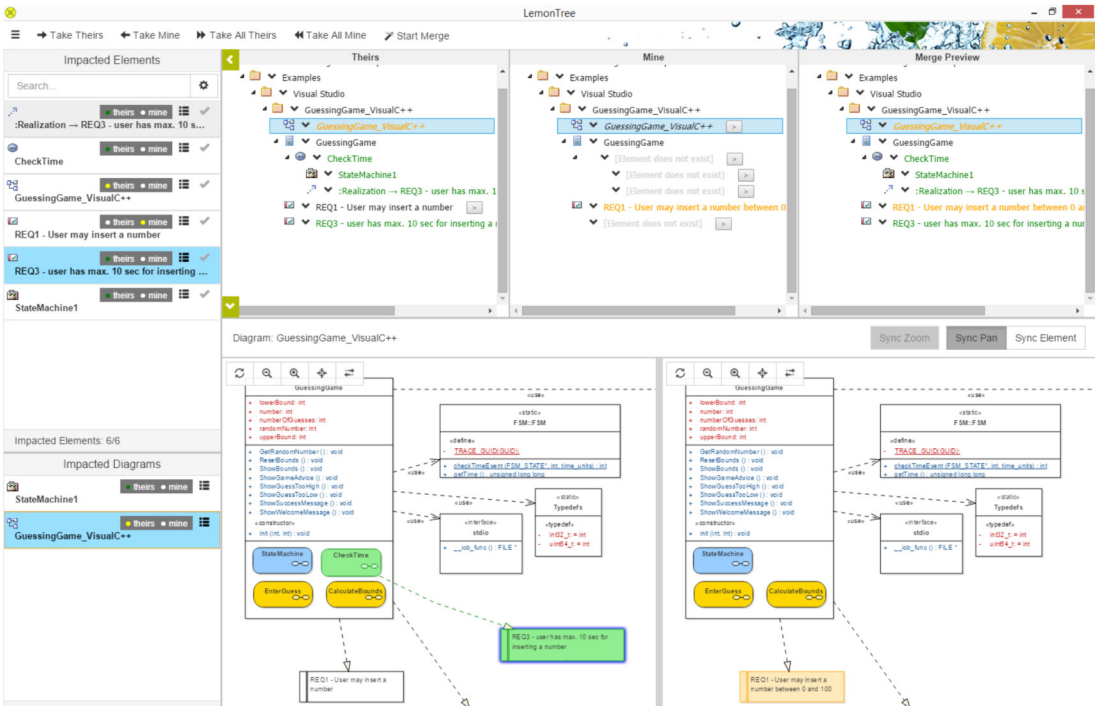


Figure 3.3: Merge preview with LemonTree [Wie17, p. 32]

LemonTree uses a 3-way diff in order to be able to display both own and third-party changes to the common base model (main version). All changes that were made are always traced back and compared with the last common main version. In contrast, a 2-way diff would only show the differences between two versions and would not provide information about who made which changes.

An example of the branching and merging concept of models is depicted in Figure 3.4. It shows the joint work on several model components and the creation of a specific model version A2, which is composed of three different model revisions. For this purpose the main version A1 is cloned (branched) and further developed by two teams (Feature Team & Basis Team). Intermediate results are checked in regularly to the respective branch. Version A2 can ultimately be created by merging all desired development results.

4 LieberLieber Software GmbH - https://www.lieberlieber.com/lemontree/

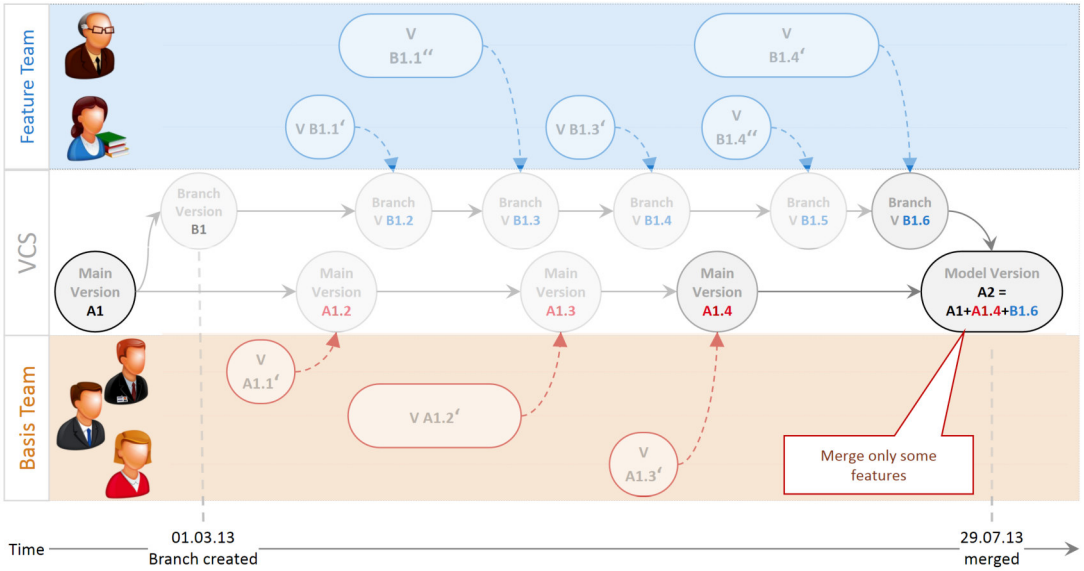


Figure 3.4: Branching and merging concept of LemonTree [Wie17, p. 38]

In order to avoid conflicts at all, there are also other approaches that prevent users from accessing the components currently under development. But, instead of these rather pessimistic approaches, LemonTree encourages parallel work. The resulting conflicts must be resolved during merging, which requires a greater knowledge of the system, but joint and distributed work yields results much faster. Ultimately, the fact that developers are constantly confronted with the changes made by their colleagues contributes to a common understanding and greater knowledge of the system.

3.2.2 PRODUCT LINE (90%)

Product lines consist of both domain and application assets. An essential aspect, however, is the development of a common asset base, containing the reusable parts. Contrary to the platform strategy, special assets can also be part of this base.

Systematic use. Since the asset base is the foundation for each product variant, its structure and continuous further development is the core of each product line. Changes, affecting both domain and application assets, lead to either modified or new assets. New variation points can therefore arise in both. Common (domain) assets must be deliberately designed for reuse when they are created. The most important aspect in this regard is modularization. This requires clear boundaries, objectives and interfaces. Furthermore, the knowledge about them must be made explicit so that they can be considered and used accordingly during application development.

Benefits. The development of a large number of reusable components offers many advantages. First of all, their use usually leads to a lean and well-structured model. As a result of this clear structure, the impact of changes can be identified rather quickly. However, the transparent separation of interfaces, components and responsibilities not only improves the representation, but also increases the understanding of all participants involved. Essentially, the use of common assets contributes to the development and allows shorter development cycles. In addition, the development risk can

be reduced because the high amount of reused components allows to reach a high test coverage. As the approach also offers the possibility of integrating application-specific solutions whose reusability does not have to be taken into account, individual solutions can also be implemented with little effort. The separation between application and domain engineering contributes to the fact that features and product variants can be developed in parallel.

Limitations. One of the biggest limitations is the high effort, especially in the initial setup, continuous maintenance and further development of the common platform. The decision to implement certain features as domain assets has far-reaching consequences and thus requires planning. The use of domain assets can also limit the degree of freedom in the development. Ultimately, this can prevent innovation. Continuous (and exclusive) reuse of assets leaves little room for truly new solutions or alternative approaches [Kru17]. Another aspect to consider is the search and identification of suitable assets.

3.2.3 PRODUCTION LINE (150%)

The use of an over-exposed (150%) model is probably the most common method to manage variants in the MBSE context. Several studies have already demonstrated how it has successfully been used to establish a product line in a professional environment. In fact, all the practical approaches examined followed this method. This circumstance certainly benefits from the fact that there is already very good tool support for it, and even existing models are suitable for introducing this method afterwards [RCC13].

Systematic use. An existing product line can be converted into a 150% model with reasonable effort. For this purpose, the variant with the largest possible proportion of components is taken and used as the new base model. All variation points (at which other system variants may vary) are then added to this base model. In this way, all components of the system variants are integrated into the base model. The variation points must be documented and described accordingly. A separate feature model represents all selection and configuration options for them. All future changes and additions will only be made to the base model. The variation points are visible and explicit, which simplifies the identification in the model. The approach is therefore very beneficial especially on a high level of abstraction.

Benefits. The creation of a base model requires relatively little initial effort. Even when setting up a new production line, the first step is to start with a variant and then gradually add more variation points. The activity is therefore initially no different from setting up a single system. Previous work can usually be transferred completely and used as a starting point for the base model. Since all variants are derived from this base model, they automatically receive all changes and no (manual) transfers are necessary. When planning a new system variant, the clear representation of features and related components is especially helpful. Similar to feature trees or OVM, technological dependencies can be traced to the realization of variants. In other words, a relationship between feature and implementation remains visible and preserved. Although the base model contains much more information, and therefore much more elements, about the entire product line than the other methods (actually all of them!),

the complexity can be slightly reduced by structuring all dependencies in a separate feature model.

Limitations. Collaborative work on a single, large model has proven to be very difficult. Concurrent changes lead to conflicts and modifications can have far-reaching effects on the entire system and all its variants. Therefore a very large testing effort is necessary to guarantee the quality for the whole production line permanently. Even variants not directly affected by a change may have to be reviewed again. Furthermore, as the number of variation points and their descriptions increases, the base model becomes increasingly confusing and complex. The derived product variants (100%) no longer contain information about variation points, this information is only available in the base model.

Example. Figure 3.5 shows the representation of a variation point within a 150% model. To demonstrate this, a rather simple state machine was chosen. In the interaction of Enterprise Architect (EA) and pure::variants (p::v) a variation point is defined as a special element *pvRestriction* of the type «constraint». In this example the variation point is named *Warnings* and affects two elements overall. On the one hand a new state (CheckValues) is added, if the feature *Warnings* is selected. On the other hand the direct transition from the state ReadSensors to UpdateDisplay is only included if *Warnings* is not selected.

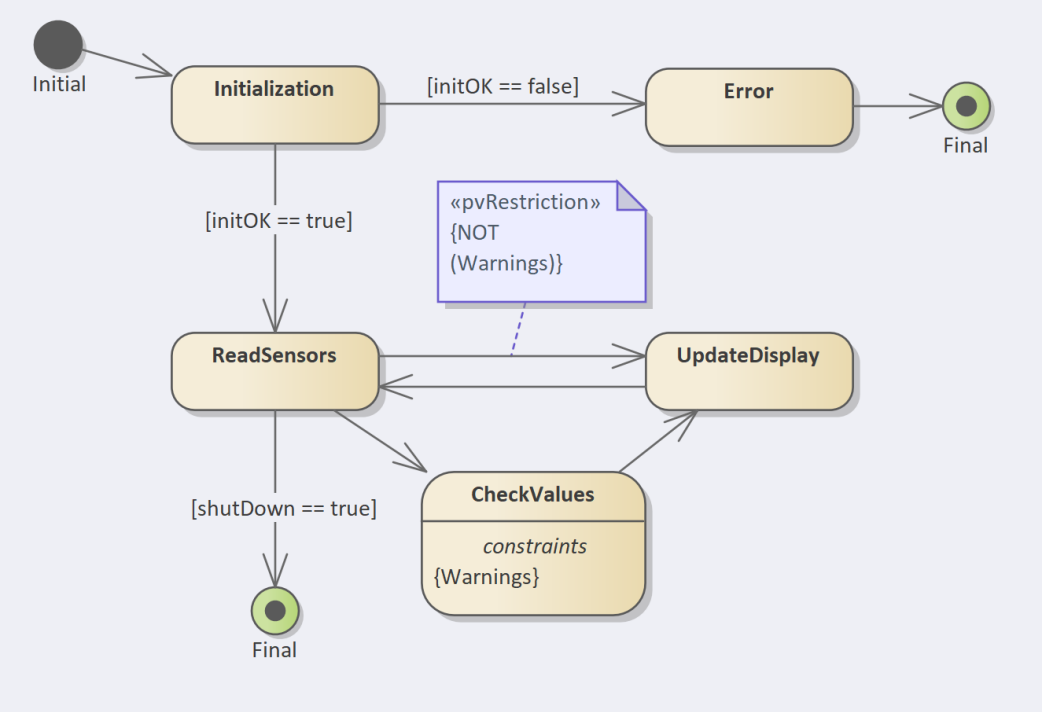


Figure 3.5: Annotation with pure::variants (example based on [Beu13])

Both possibilities to attach a variation point to an element are shown in this example. They can either be defined internally or attached externally as a note. A corresponding feature model in p::v allows to include the variation point *Warnings* as an alternative/option in a specific product variant and to automatically create a 100% model of the variant, which only contains the selected elements.

DECISION SUPPORT

A short summary of the points discussed is given below. For this purpose, Figure 3.6 shows the connections between important criteria and their relationship to one of the three approaches depicted. The lines intentionally do not indicate a direction. Thus the image can be read in both directions. A single criteria has **positive** effects on linked approaches, just as an approach **benefits** from all linked criteria.

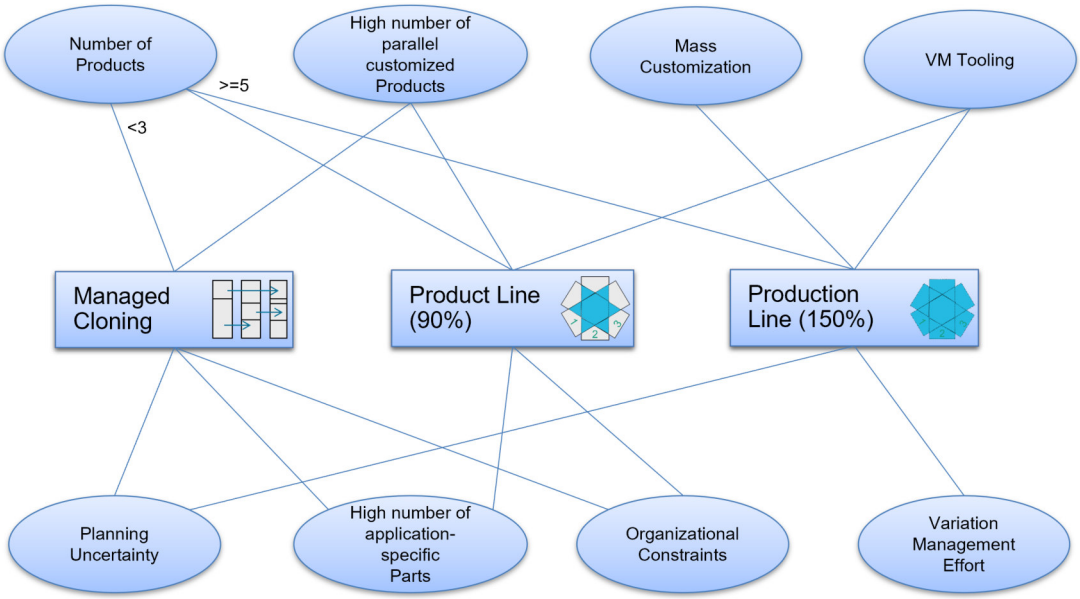


Figure 3.6: Decision support for variation management

Number of products. As the number of products increases, so does the complexity and obscurity of their variants. In order to keep track and gain as much benefit as possible from the reuse of their components, product or production lines are therefore advisable. Managed cloning should only be considered if the number of products is low.

Parallel Customization. If many variants have to be developed and created in parallel, working on a single common basis is a limiting factor. Therefore, product lines or managed cloning are more suitable for this purpose, whereas a production line can be quite difficult to maintain.

Mass Customization. Customer-specific mass production can only be effectively realized with a production line. All variation points and their characteristics are fully described, which enables the fast and automatic generation of variants.

VM Tooling. Since variant management is not part of system modeling, additional tools are required to implement product or production lines. If these tools are already available and their application is familiar, it is advisable to utilize their support and set up a product or production line.

Planning Uncertainty. If there is a high degree of planning uncertainty, the use of product lines should be avoided. This is the case, for example, if variation points

often have to be inserted or changed subsequently, or if there are uncertainties in the separation of special and general components. In a production line there is no separation between domain and application assets, so variation points can be integrated even late in the process.

Application-specific Parts. As the number of application-specific parts increases, the significance of production lines is declining. The integration of too many special parts in a 150% model is not reasonable, because of their low reuse. In this case, product lines that can be extended according to customer specifications are a good solution.

Organizational Constraints. Organizational constraints can result from distributed teams, for instance. These may be divided according to customers, functions or responsibilities, making it difficult to work together on a common model. Therefore, these constraints primarily have a negative effect on production lines, where a single base model has to be maintained. Merging changes, for instance, can be difficult once time differences or language barriers have to be overcome.

Variation Management Effort. While the effort for managed cloning hardly changes for each new variant, it does decrease for a product line by the proportion of reused components. On the other hand, once a production line is set up, the generation of variants requires the least effort. If the goal is to be able to produce many variants with minimal effort, then the production line is the best choice.

Although this list is certainly not exhaustive, it gives an idea of aspects to consider when choosing a suitable variant management approach. On the other hand, it also helps to identify limiting factors to avoid selecting a method that may be unfavorable. In this aspect, it refers to RQ3, along with the limitations of tool support, which are described in more detail in the next section.

3.3 TOOL CAPABILITIES / LIMITATIONS

Motivated by the examples from the previous section, this section aims to discuss the necessary/desired capabilities and limiting factors of tools in more detail.

All previously described approaches require the support of tools for an efficient and systematic application. Depending on the chosen approach, there are sometimes identical, but sometimes also very specific requirements for these tools. The following overview in Table 3.2 is intended to give an overview of some of these requirements. If a specific approach is to be used, a tool that meets the necessary requirements should be chosen if possible. However, a suitable method can also be identified based on the capabilities of a tool already in use.

CAPABILITY	MANAGED CLONING	PRODUCT LINE (90%)	PRODUCTION LINE (150%)
Annotation	○	◐	●
Automatic generation of variants (100%)	○	◐	●
Validation (REQUIRES, XOR) of feature selection / combination	○	○	◐
Branching (creation of a new variant / product)	●	◐	○
Merging (conflict indication & resolution)	●	◐	◐
Comparing (2/3-way diff and preview)	●	◐	◐
Cherry picking (patch, without rebase)	◐	○	○
Version control (model history, documentation, traceability)	●	◐	◐
Impact analysis (on other variants)	○	◐	◐
Review / approval step	◐	◐	◐
Import / export	○	●	◐
Asset management (search & find, identification of common assets)	◐	●	○

● = required ◐ = optional ○ = not necessary

Table 3.2: Overview of tool capabilities

The list does not claim to be complete, but merely serves as an overview. The absence of tool support for a particular task does not automatically mean that the corresponding method cannot be used. However, the task may then have to be executed manually, which is both more time-consuming and error-prone. Especially if it concerns a frequent repetition of certain tasks.

In practice, the listed capabilities are often not bound to a single tool, but can/must be performed by several interacting tools. Further restrictions result from this (often multi-vendor) distribution. With annotation, for example, the modeling tool must support some kind of annotation in the model, whereas an additional selection tool must be able to read and interpret this annotation. Standard interfaces such as VEL can be used for the exchange of information, or the tools are already designed by the manufacturer to be compatible. A widely used combination of the tools Enterprise Architect (EA) and pure::variants (p::v) for the realization of variability in a 150% model, which was also used for the feasibility study, will be examined more closely for possible problems at the end of Section 4.2.

3.4 VARIABILITY REALIZATION MECHANISMS

In the following section, mechanisms for the realization of variability are listed and explained. They were selected because of their applicability in the MBSE context and thus to models. The mechanisms will be characterized according to their underlying principle, advantages and limitations. Wherever possible, they are supplemented by small examples that are intended to demonstrate their practical use.

The findings of Zhang et al. were used as a starting point for the list of mechanisms [ZDB16]. They have been transferred to the model level and it has been checked whether the original statements are still valid for models. In order to increase the understandability, the terminology was adapted accordingly. The mechanisms presented are intended to answer RQ1.

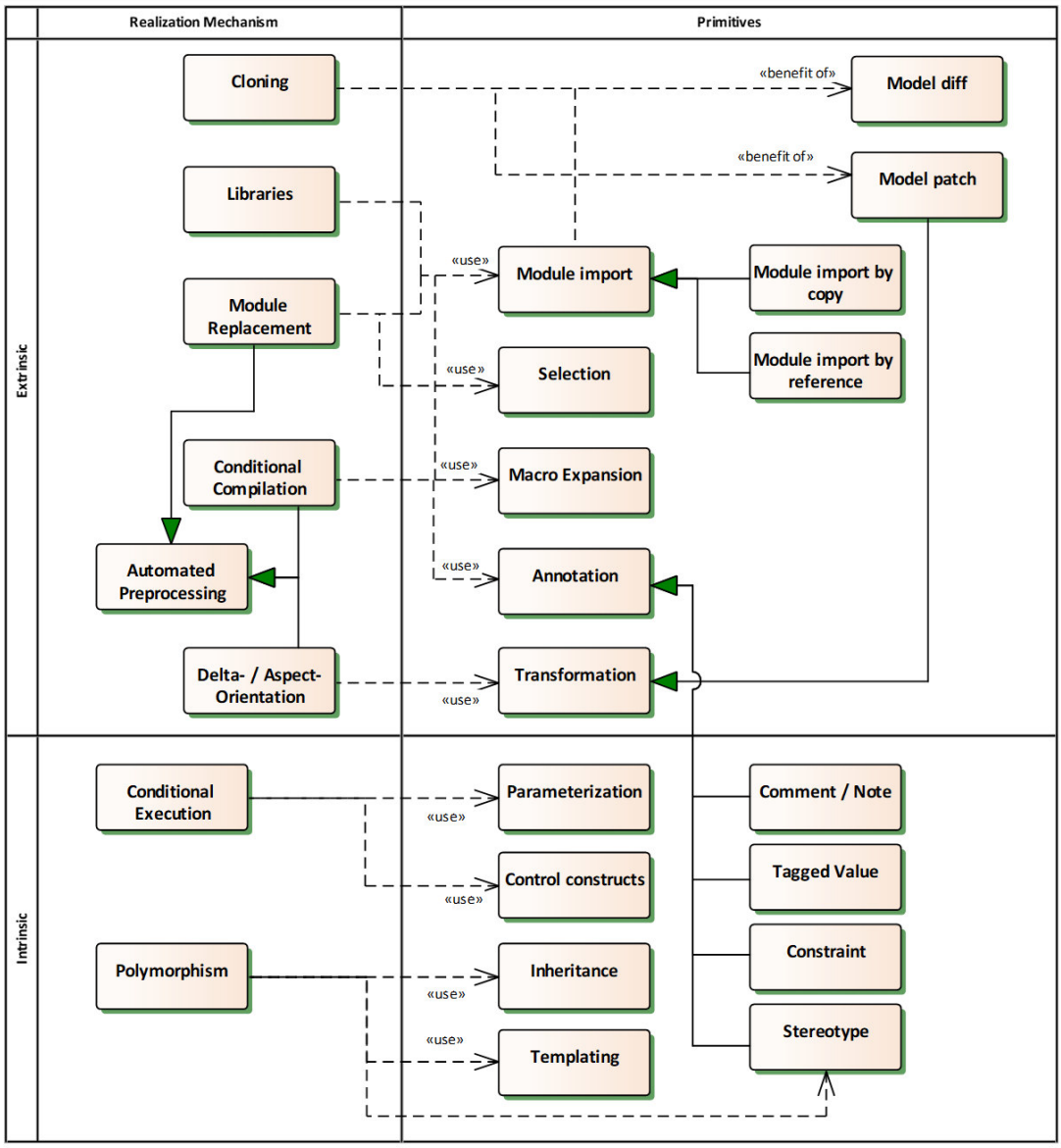


Figure 3.7: Overview of Variability Realization Mechanisms

The illustration in Figure 3.7 provides a first overview of the identified mechanisms and their underlying principles/primitives. On the vertical axis the mechanisms have been classified into ones that are extrinsic to the modeling language (SysML) and ones that are intrinsic, i.e. they use features of the modeling language itself. Table 3.3 briefly characterizes the used primitives in more detail.

PRIMITIVE	DESCRIPTION
Model diff	Comparison and visualization of differences (added, removed, modified) between models, i.e. their views, elements and values.
Model patch	Utilizing a revision history, certain modifications can be packaged in a patch and subsequently applied (merged/transferred) to other models.
Module import	Complete or partial insertion of own or prefabricated (third-party) model parts.
Module import by reference	Only a reference to the prefabricated model element is added. When changes are made to the source element, they are automatically transferred to the variants. This can be an advantage in terms of maintainability, traceability and consistency, but also a disadvantage in terms of ambiguity of all side effects, their testability and system-wide plausibility.
Module import by copy	A copy of the prefabricated model element is created and added. This copy may evolve independently and has no relation to the original. Disadvantages result from the high maintenance effort and the low reusability.
Selection	Mechanism to select/deselect variant elements based on decision criteria, e.g. if/switch constructs.
Macro Expansion	A preprocessor substitutes macro with predefined content, e.g. replacement of a parameter by a specific value or simple calculations when deriving a specialized model instance.
Annotation	Embedding variability constructs formulated in another language into the modeling language.
Stereotype / Tagged Value	Extensibility mechanisms for creating new model elements by introducing domain/problem-specific properties. Usually, these are derived from existing classes or elements but can be adopted for specialized usage.
Constraint	Used primarily to model physical system parameters and restrictions, such as performance or reliability, by adding mathematical or logical expressions. User-defined constraints can also be used to define variation points (especially with p::v).
Comment / Note	Textual annotations, usually intended for human readers to increase clarity on certain aspects.
Transformation	Automatic generation of a model, using a combination of composition, selection, and generation mechanisms or by successive application of multiple model patches.
Parameterization	Introduction of characteristics, whose state is evaluated at runtime and used for control flow decisions, behavior control or component selection.
Control constructs	Elements and keywords provided by SysML that facilitate a variable control flow decided at evaluation time.
Inheritance	Specialization (by adoption, modification, extension) of existing properties and methods of a more general element. Multiple elements of the same type can exist and the selection can occur at runtime.
Templating	Ability to predefine structures generically and specifically equip them when used in UML.

Table 3.3: Explanation of mechanism primitives

CLONING

Cloning is a very simple and intuitive approach to create variants in all engineering disciplines. The cloning approach can be used in any phase of the system development process and offers the possibility to quickly create a new and independent system variant.

Principle. The new system variant is created by copying an existing system model and then filling it with the new specific system properties.

Use. Since the size of the cloned system model is not important, it can range from a single artifact, e.g. a block, functional structures and subsystems to complete systems. Technically speaking, a clone can be created either by copying the artifacts directly or by using configuration management branches (cf. managed cloning).

Example. Figure 3.8 shows the generation of two product variants using cloning. Model variant 1 (left) shows the decomposition of a car into 3 blocks, of which the combustion engine system is described in a generic way. The model variants 2 and 3 are now created by copying the entire model and replacing the generic engine block with one that is specific to its variant. Both new variants are independent and can implement their own behavior. Although it is not necessary to generate and/or maintain a generic variant, it helps to indicate the variation point.

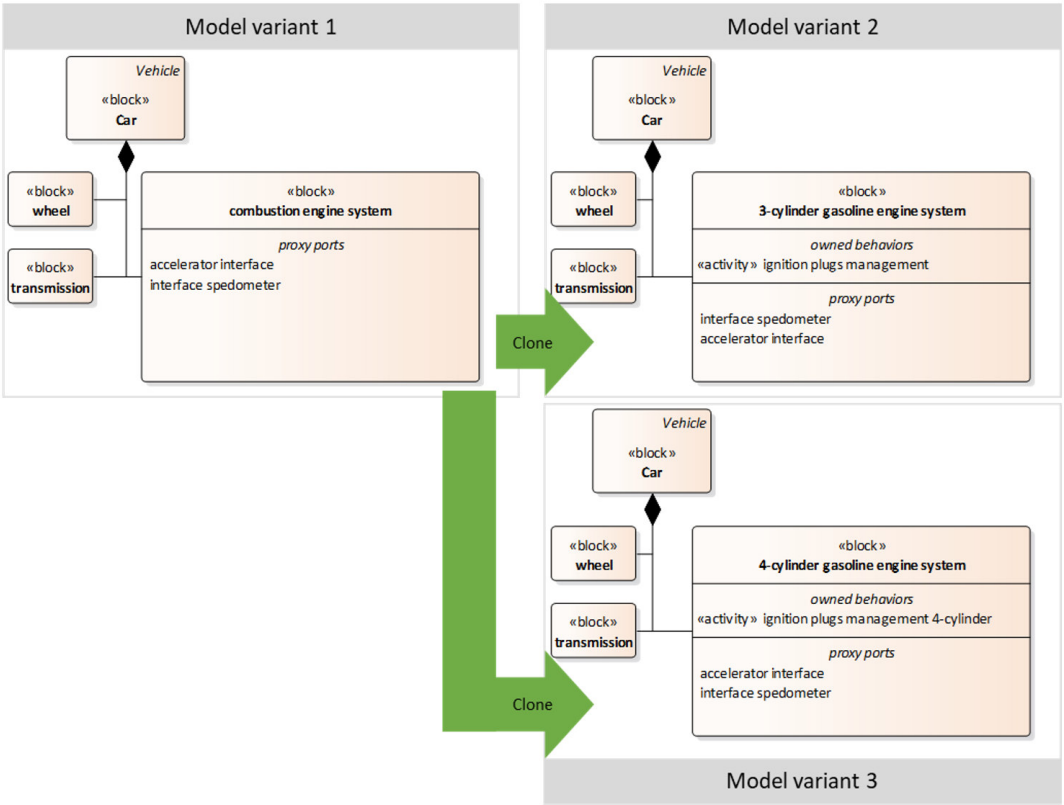


Figure 3.8: Cloning

Advantages. The cloning approach can be applied regardless of the modeling language, method, and tool. This makes it fast and quite cost effective, especially in the early stages of creation, which is further supported by the fact that no special knowledge of the approach itself is required. The maintenance and evolution of the newly created system variant is furthermore independent from its source, i.e. changes in the respective system variants have no influence on each other. Thus, cloning does not pose a risk for existing (and possibly verified) system variants - in contrast to most other approaches. Changes do not have to be coordinated with other system variants, which again influences speed and simplicity. In the context of prototyping and early testing, the properties of the cloning approach described above are a big advantage. A new system variant can be quickly and easily checked for feasibility and, if necessary, quickly discarded due to its independence. Only a small amount of development effort is required [KG08].

Restrictions. Since there are no direct dependencies between the model clones, all changes are only local and must be transferred manually between the system variants as required (for example, feature update/upgrade). As the number of variants increases, this leads to a lower maintainability of the system family as a whole. In addition, modelers and users of the model variants cannot see which parts of a model have been copied and which changes may need to be applied to other variants as well. The knowledge of similarities between the clones tends to be quickly lost – in result, the engineers might forget to port a bug fix or a functionality upgrade to some relevant clones [Dub+13]. Without appropriate tool support (e.g. a 2/3-way-diff of model variants) the comparison of model variants quickly becomes a complex and error-prone task. Although cloning is initially intuitive and simple, over time independence is usually paid for with increased maintenance costs and declining model quality. Therefore, the approach should only be used if a small number of variants (<= 5) and low frequency of changes over time can be expected. Cloning of model fragments can be also deliberately used as a variability mechanism if their later similarity is expected to be low [Tis+12]. In this case, the rationale for using Cloning is the high effort and low benefit of using other variability mechanisms such as Conditional Compilation for the dissimilar code. Deliberate cloning is therefore performed if there is no prospect of a viable variant consolidation in the future. A detailed investigation of the motivation for and experience with cloning in an industrial environment is provided in the study by Dubinsky et al. [Dub+13].

LIBRARIES

The use of libraries is widespread in all engineering disciplines. A typical application of this in MBSE is the field of standards. In this context, standards in static architectures are already provided for use, such as ISO/IEC 80000 for the definition of value properties.

Principle. Common model elements are stored in libraries, which can then be used to create the system model.

Use. Prefabricated (third-party) libraries are integrated on a functional, logical or physical level and their content is used by the other system packages. This principle

is well-known from other disciplines like software and hardware. In software engineering, libraries can be linked statically or dynamically to the rest of the system. In hardware development, third-party CAD models can be provided to facilitate the integration of the respective hardware parts. For example, 3D models with exact mechanical dimensions are available that can also describe the electrical behavior of the components under various conditions and enable their simulation. Libraries form rather independent units and contain functions and properties that can be called or used by other elements. When creating libraries, the focus can be on simple and direct reuse or the possibility of providing elements, systems, functions without having to provide an insight into the internal structure ("black box"). The use of libraries requires well-defined interfaces and highly isolated components. It is therefore a method that supports modularization. When library functions are used in multiple locations, the risk of topological variability increases. However, especially in the modeling context, libraries can also be used to manage user-defined elements. SysML allows the creation of stereotypes that can be used for recurring design decisions. A systematic use of design libraries was introduced by Kruse. For this purpose several libraries have been set up to support functional, behavioral and structural modeling. All of them provided generic SysML elements for reuse [Kru17]. In this aspect it resembles Module Replacement.

Advantages. Low engineering effort in combination with high reuse rate. If the library can be locked for modifications, this guarantees the consistency of library usage across all variants. The use of libraries thus contributes to a clearer structure and separation of responsibilities. Parallel working is (partially) supported.

Restrictions. Often libraries are only available in compiled form as a "black box". The concrete inner life is not known and cannot be changed. Due to this unchangeability, errors are very difficult to reproduce and cannot be corrected independently. If there are changes (e.g. due to updates) the effects and possible undesired side effects on other model components are difficult to overlook. The extent of the components in a library is limited to modules/functions. Depending on the design of the library, there may be little or no possibilities for parameterization, which can have a major influence on the realization. If, for example, a different representation of an output variable is required and the library offers no possibilities for this, the only remaining option is the often time-consuming search for another, more suitable library or an adaptation by means of wrappers (cf. Adapter Pattern [Gam+95]). Changes at the interface of a library, also leads to high efforts.

MODULE REPLACEMENT

A widely used approach to modularization and realization of reusable components in both mechanical and software engineering.

Principle. The models are modularized into submodels with a uniform interface. If the system variants differ in individual modules, the respective module variant is created and integrated into the new system variant by a selection mechanism, e.g. the selection of different state machines or processor modules (SOM) in different system variants. The approach is in principle very similar to Libraries. However, libraries have

a stricter organization and also refer to prefabricated third-party components whose implementation is unknown. Both mechanisms follow the substitution principle and require therefore well-defined interfaces.

Use. This principle is known from Product Lifecycle Management (PLM) systems, for example, where part variants can be selected in generic parts lists by feature code or parameters. The decisive factor here is the use of a uniform interface. In hardware development, this can be realized by using standardized connectors, for example. In software development, this principle can be found, for example, in the selective binding of software libraries or components at compile/link/runtime. With regard to MBSE, the VAMOS approach also follows this principle [Wei16], but is limited to the visualization aspect. A matrix view allows the identification of the modules belonging to a selected variant. However, it is not easily possible to generate an automated model view from it.

Example. Figure 3.9 shows the transformation of a model with the generic block "combustion engine system" (left) into a specialized model in which this block is replaced by one of the «ModelLibrary» variants. It is important to note that any of the three available variants in the library can be selected. Although their internal behavior may differ, all variants share a common interface and are therefore interchangeable.

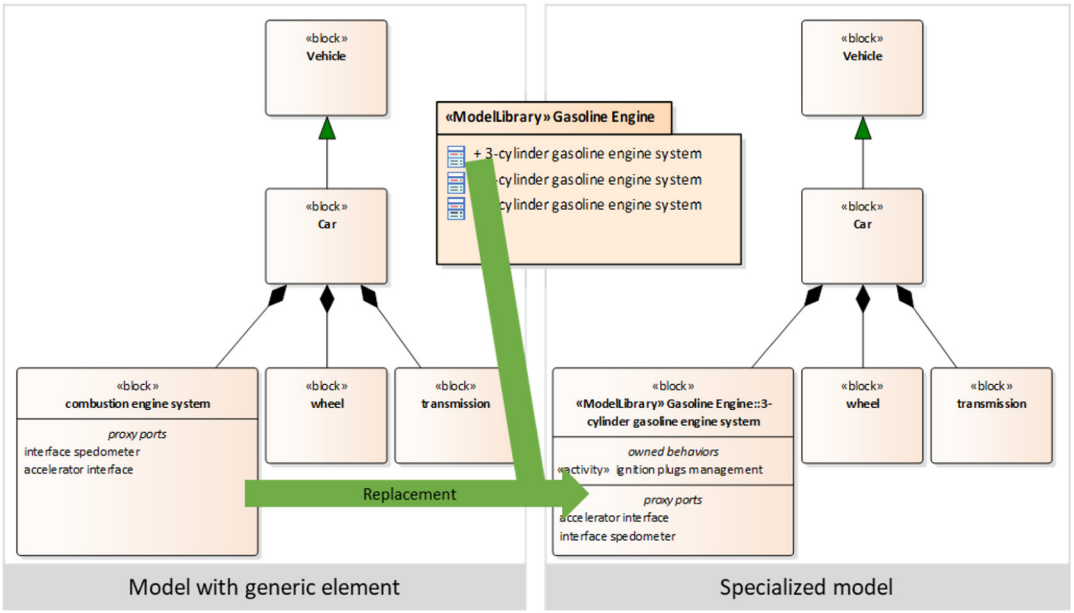


Figure 3.9: Module Replacement

Advantages. The approach is rather easy to implement. If the modules are selected manually, no special tool is required. Maintaining clear interfaces between the model modules/packages promotes the basic idea of strict separation of concerns. Generated modules thus have a high potential for reuse.

Restrictions. In order to enable an automatic creation of individual system variants, the selected modeling tool must support appropriate modularization and model import capabilities. Cross-module dependencies and restrictions must usually be

described symbolically, since the elements in the module variants often have different IDs. If changes are made to the module interfaces, these must be transferred to all variants. Unlike with libraries, the variants here exist as separate components having identical interfaces but do not share one common interface.

CONDITIONAL COMPILATION (AKA. ANNOTATION)

In the context of feature-based systems and software product line engineering conditional compilation is a well-known and widely adopted variability realization mechanism.

Principle. In a so-called 150% model, the entire system family is modeled in a generic manner. The respective model variants (100% models) can be automatically derived from this 150% model via parameterization and a selection mechanism.

Use. The approach follows the preprocessor principle, i.e. special annotations are used to identify the model parts that are not contained in all model variants. A separate preprocessor tool is then used to make the necessary adjustments, typically removal of optional elements, parameterization, or the selection of alternatives. Tools such as pure::variants (p::v) or Gears[5] support this automatic derivation of variants. They also allow to define constraints and dependencies between selections. SysML itself includes means to annotate elements such as comments, tagged values and constraints to provide additional information about these elements and to allow data transfer to the preprocessor tools.

Example. The user interface of p::v depicted in Figure 3.10 shows on the left side a list of selectable features. The right side shows an exemplary section of the corresponding 150% model. As can be seen in this example, the optional feature "*acceleration_to_speed*" is not selected and corresponding elements in the model are therefore grayed out. Once all features for a variant have been selected, the model for this variant can be generated automatically. All grayed out elements will not be included.

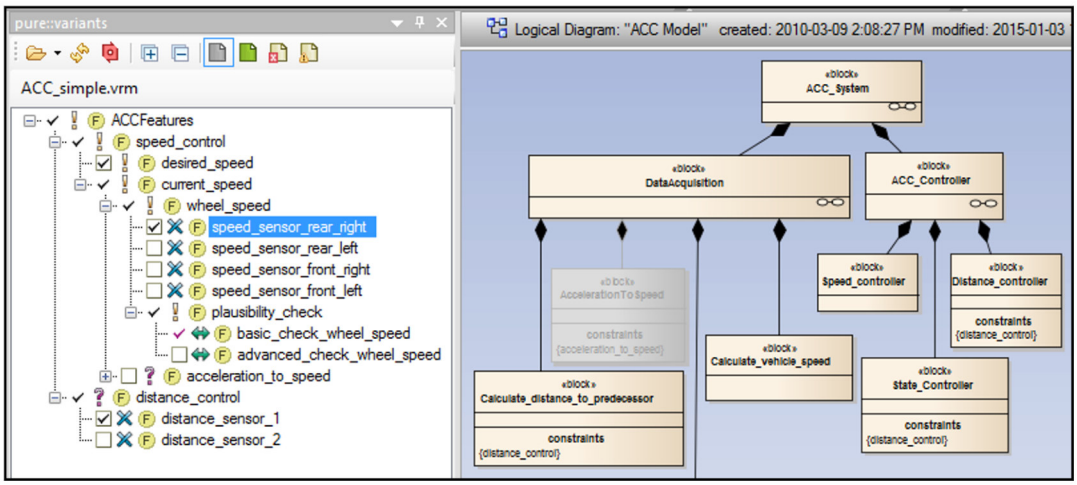


Figure 3.10: Annotation [DAB15]

5 BigLever Software, Inc. - https://biglever.com/

Advantages. As changes are only made to the 150% model, all model elements are automatically reused. Furthermore, feature updates/upgrades can be applied very quickly to the different system variants. Changes are made directly in the 150% model and, if necessary, are selectively activated via parameters for the system variants. The preprocessor tools often also support a comparison of the system variants with each other. In this way, differences between system variants can be quickly identified and reviewed.

Restrictions. After the preprocessor has generated the model variants, the original variation points are no longer identifiable, i.e. it is no longer clear in the derived model, which parts of the model are also used in other model variants or how the model variants differ. Both dependencies and parameterization can only be handled at the more abstract configuration level and are often not visible in the model itself. As the number of variable model elements increases, the complexity of the corresponding views increases as well, since the 150% model contains the full range of all possible configurations. If there are many optional elements, e.g. blocks/ports or states, or if the model elements are connected in different ways, the resulting views quickly become confusing. Analogous to the software world, this can quickly lead to a preprocessor hell, where the core content of the model is no longer directly apparent due to all the preprocessor annotations. When working on a common 150% model, problems often arise in everyday practice due to concurrent development activities in the different system variants. If the model is changed for one system variant, it must be ensured for the remaining variants that this change does not cause any unintended side effects. Obviously, this leads to additional coordination and assurance efforts.

DELTA-/ASPECT-ORIENTATION

Principle. Delta-/Aspect-Oriented approaches strictly separate the core from change set (aka. aspects, delta), which can be automatically applied on demand onto the core.

Use. As this mechanism is typically not supported by modeling languages, it requires an external transformation tool. Similar technology-centric approaches have been repeatedly proposed as Subject-Orientation, Feature-Orientation, Change-Orientation, and Delta-Orientation.

Advantages. As a compositional mechanism like Module Replacement or Polymorphism, Delta-Aspect-Orientation has in theory similar benefits such as clear separation of common and variant content. Any model element can become subject to modifications injected into the model from the outside. Furthermore, Deltas/Aspects provide means of consolidating variant behavior, which would otherwise be scattered over many modules and tangled with common content as well as with other variations. This might lead to better model maintainability.

Restrictions. As Delta-/Aspect-Orientation is normally not supported by a modeling language itself, it is difficult to be applied rapidly in engineering and probably needs more learning effort. Furthermore, it is often not that trivial to oversee the full impact of a change. These are probably reasons, why these mechanisms are not widely used in industry.

CONDITIONAL EXECUTION

Models are evaluated by both humans and machines in order to convey and document system properties and behavior. As with programming languages, some model contents can be subject to conditions, i.e. they are only considered in certain situations or configurations.

Principle. At evaluation time of the model the decision is made on whether and how certain generic model elements, such as optional blocks or parameter values, should be used.

Use. In SysML there are means to model optional or alternative behavior, for instance decision nodes in activity diagrams and conditional function calls. Within sequence diagrams, variation in system interaction can be modeled by using keywords such as alt, par, or loop. The assurance diagram offers another possibility to model parametric relationships between properties of system elements [Wei14]. The decision criteria can stem from software parameters, hardware configurations, or user interactions. Every state the system can take and every decision path is defined during the modeling of the system; the actual use is chosen and evaluated at execution (view) time.

Advantages. The model retains a high degree of flexibility due to the late binding to variants and thus offers easy adaptability to special needs, as all relevant information is summarized in it. A further advantage of the Conditional Execution is the simple and intuitive application. The impact of the variability can be analyzed with analysis tools that come along the modeling language. In addition, the late instantiation of the model that comes with this approach offers good opportunities to facilitate the management of unforeseen requirements.

Restrictions. As the number of variants increases and the lifecycle progresses, a clear and unambiguous distinction between the common parts of the model and the specific ones becomes more and more difficult, as there is no direct separation of. A particular strength is to be found in the decision criteria of the different model levels. This and the easy adaptability lead to a very low reusability of the general parts. This approach is therefore more suitable for systems with a small number of variations, which are also limited to a small part.

POLYMORPHISM

A widely used principle in software development is polymorphism, which can be considered as the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

Principle. Functions and classes exist in several different specializations (types), whereby the selection of a specialization can take place either at the time of modeling or during the evaluation of the system variant.

Use. The use of a generalized element makes it possible to maintain a uniform interface for all its instances. It furthermore allows the inheritance of functions and attributes from the generalized element to the specialized element. The ability to select a specific type during instantiation allows to create dynamically systems. This method is

already widely used in software development and part of various design patterns. The languages SysML/UML support inheritance through their generalization/specialization relationships as a language construct.

Example. In Figure 3.11, the combustion engine system is first divided into the two blocks diesel and gasoline. For the gasoline variant there are now again three alternative types depicted. With each level, additional functions and properties can be defined - the elements are thus described in an increasingly specific form. However, each higher-level element also defines more general behavior, which is common to all special element types. Although the representation is resembling feature trees, the focus here is not on the organization and classification of features but on their implementation. It can nevertheless help to visualize variation points.

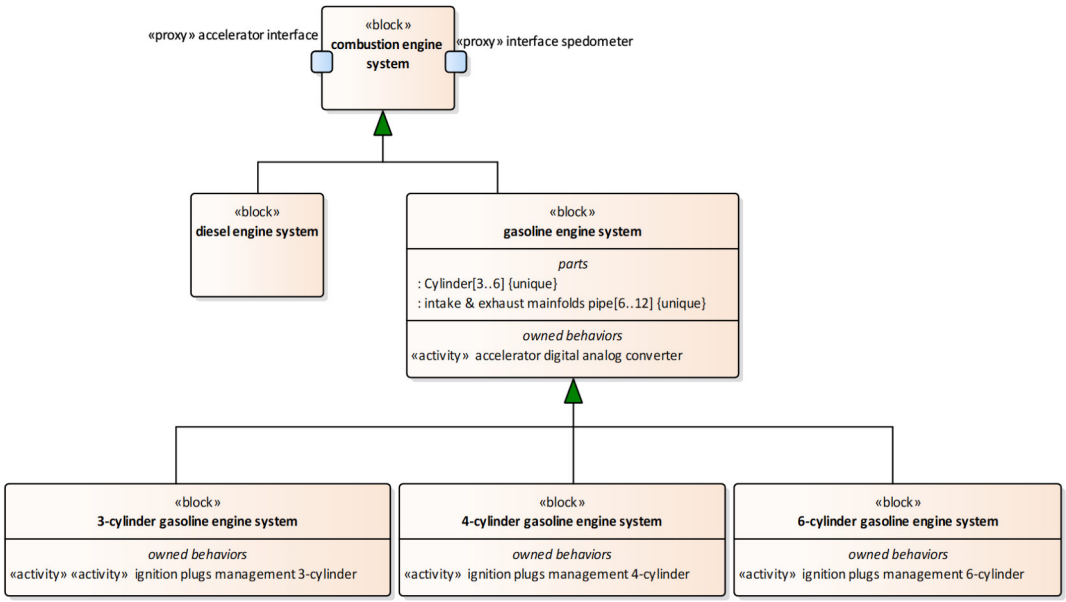


Figure 3.11: Polymorphism

Advantages. By defining a generalized element and the resulting reduced effort to create a new specialization, a simple and fast mechanism is provided to specify new variants. System interfaces can be designed as generalized elements, allowing plug-and-play functionality. Furthermore, generalized elements offer high reusability and can be easily extended.

Restrictions. Extensive usage of polymorphism tends to increasing the complexity of the models, especially when inclusion polymorphism is used. Changes to existing interfaces are cumbersome and error-prone due to their far-reaching effect on all variants (children). If type selection is dynamic at evaluation time, this selection may not be deterministic and may be difficult to reproduce.

3.5 KEY PERFORMANCE INDICATORS

In order to facilitate a conscious decision for or against a variability mechanism, relevant criteria that should be considered in the decision process are discussed below. Of course, such decisions cannot be made universally and completely detached from the system context. Nevertheless, this list of criteria is intended to show, with reference to the RQ2, how the identified mechanisms can be characterized and assessed.

Method. As mentioned earlier, not every mechanism is equally suitable for every variation management approach. While modularization is the main focus for the development of a product line (90%), the production line (150%) rather requires possibilities to label optional elements. However, the association to a method cannot be understood as binding. Rather, it is an indication of which mechanisms can be regarded as more helpful once a particular method has been chosen.

Tool support. Although this aspect should not be overestimated, tool support may be necessary for the efficient application of a mechanism. If the modeling language already contains all necessary features for this purpose, this ultimately saves time and effort for the acquisition, maintenance and training of additional tools.

Effort. Here two aspects must be taken into account. On the one hand, there is an initial effort, which ultimately enables to derive a new product variant from the product line by using the selected mechanism. This includes, for example, the creation of a library or the development of a 150% model before it can be used. Some mechanisms have no initial effort, e.g. cloning. The second aspect concerns the remaining effort, which exists after all necessary preparatory work has been done. This also includes the existing experience. Especially in brownfield engineering, there is always pre-existing experience that should be considered. Sometimes it is preferable to take the second or third-best choice, if there is already experience with the respective mechanism. To this end, the following questions should be considered: How did the product or product line come into being and what initial efforts have already been made to support variability? Which mechanisms and artifacts already exist within the individual engineering disciplines or already at system level? How can these be meaningfully adapted, changed or supplemented? Which modifications develop the product line further and still retain enough flexibility for future changes? Ultimately, it is precisely in the last point that a cost-benefit comparison must be made. How much effort is justified to maintain the flexibility of a product line?

Granularity. Depending on the used variability mechanism, the granularity of the variation points and respective realization variants differ. While some mechanisms, e.g. cloning, support almost any granularity of changes in the model, other mechanisms enforce a certain size or form of the variation points within the model (e.g., a modeling element, a view). While mechanisms supporting any granularity range are more flexible in variability realization, model variants using mechanisms with a limited granularity range are usually more disciplined, well-structured, and easier to maintain. Regarding variability and quality (non-functional) attributes, Etxeberria et al. state that, in "a product line, quality attribute requirements have also variability, because not all the products require the same level of security, performance, etc. This aspect has [...] been neglected or ignored by most of the researchers as attention has been

mainly put in the variability to ensure that it is possible to get all the functionality of the products" [ESB10]. In relation to models, this is particularly reflected at the parameter level. A high granularity is generally necessary to realize such parameter variability.

Variant isolation. Depending on the used variability mechanism, realization variants for each variation point are either included in one package or stored in isolated modules or packages. Based on this difference, Kästner [Käs12] grouped these mechanisms into two categories: annotative mechanisms (i.e., Condition Compilation, Conditional Execution) and compositional mechanisms (the remaining ones). Variability realizations using annotative mechanisms are more integrated but potentially more complicated to understand, while the variability realizations using compositional mechanisms are less complicated but also more fragmented (cf. [Ape+13]).

Open variation. A related concern is whether a mechanism supports open variation, i.e. the range of possible variants for a variability can be easily extended. In consequence, the respective realization variants need to be added to the variation point. If open variation is supported, these additions can be realized without interference with the existing variant realizations. For instance, external modelers can provide customization packages (e.g. plugins) extending the model with their own model elements, while treating the core as a "black box" with defined variation points. This is especially helpful to avoid that some core properties need to be assured again and again, every time a new variant is supported for an open variability. On the other hand, open variation can become unmanageable and uncontrollable.

Explicit variation points. Depending on the used variability mechanism, the form of variation points differs. If annotation is used, variation points are annotated model fragments, which are explicitly represented and easy to identify. In contrast, if a capability of the modeling language is used for variability realization, e.g. parameterization in combination with some conditional evaluation, then it becomes hard to distinguish between product line variability and some other aspect, where parameterization is used. Since variability is a cross-cutting concern with (often highly) scattered realizations, mechanisms with implicit variation points tend to cause challenges in development, maintenance and understandability in general.

Independence. Can variants be further developed independently of each other? Through a high degree of independence, the influence of a change on other variants can be avoided. This is of particular interest if concurrent changes often have to be made to individual variants or if operational constraints divide feature development among different teams or locations. On the other hand, this can lead to considerably more effort if the same changes have to be applied to several variants in the same way.

Support of defaults. In variability realization, Patzke [Pat11] argues that the default selection in a variation point can reduce the number of variants (by one) and simplify the variation logic. It is especially convenient when features of a variation point are optional (in this case the default is null and requires no specialization at all).

An application of these mechanisms and a subsequent evaluation based on these KPIs will follow in the feasibility study.

4

FEASIBILITY STUDY

For the implementation and demonstration of feasibility, a practical example is used. BCON, the virtual company introduced in the master's thesis of Zurbuchen, is used for this purpose. The company offers both weather stations and controls for refrigerated transport. In the course of the thesis, commonalities, especially in measurement technology, were identified and the resulting product line was transferred into an integrated lifecycle management concept [Zur14]. In order to illustrate different aspects of variant realization, a few change scenarios for the product line are first introduced and subsequently applied.

4.1 INTRODUCTION

The product line evolution is exemplarily shown by some change scenarios, that were already presented in the master's thesis [Zur14]. The scenarios shown in Table 4.1 were selected because they represent typical scenarios and focus only on the manageable product line of the weather stations.

SCENARIO	DESCRIPTION
(2)	*WeatherStationConnectPro* shall add the influence of wind to calculate real outdoor temperature.
(4)	*WeatherStationConnectTrendline* and *WeatherStationConnectPro* shall have an ozone sensor to give advice for outdoor sports.
(6)	The temperature measurement calculation is wrong in all products.
(12)	*WeatherStationConnectPro* shall display battery status of connected sensors.
(13)	*WeatherStationConnectPro* shall issue a (temperature) warning when temperature is falling or rising above certain temperatures or any other simple condition on any sensor value.
(17)	*WeatherStationConnectPro* shall have a new feature called RoomHealth, which monitors and controls the temperature and humidity in a room via its home automation interface.

Table 4.1: BCON: Product evolution based on [Zur14, pp. 55-56]

The scenarios listed in Table 4.2 were added in order to further elaborate and illustrate certain aspects.

SCENARIO	DESCRIPTION
(20)	*WeatherStationConnectBasic* shall be able to operate at ambient temperatures of -50 degrees Celsius.
(21)	*WeatherStationConnectTrendline* shall support digital temperature sensors.
(22)	*WeatherStationConnectTrendline* shall support a digital high precision temperature sensor.

Table 4.2: BCON: Additional product evolution

To increase readability and provide a quicker overview of progression, the scenarios are shown graphically in Figure 4.1. The representation focuses exclusively on product features and shows their evolution over time. To this end, the features of all individual products are first shown in their basic configuration (left). This information was mainly retrieved from the product feature matrix (cf. [Zur14, pp. 126-127]). All further product variants can be derived from already known feature sets by reusing, adding and changing these features. At the time of release, each product has a unique combination of name and version identifier. This is represented as a box.

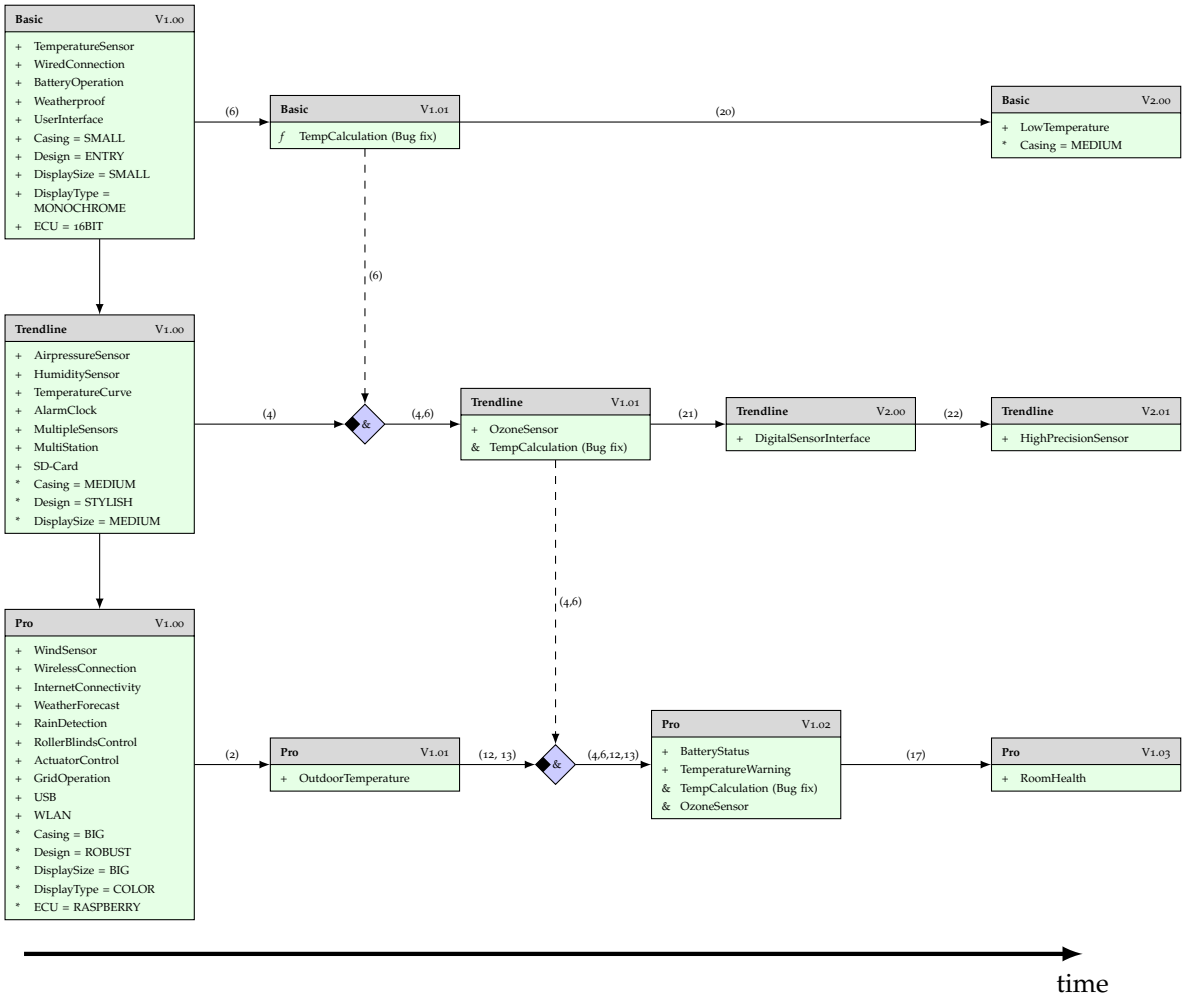


Figure 4.1: BCON: Visualization of product evolution

The only relationship between nodes is inheritance. All properties are automatically propagated. Each node therefore only needs to specify its changes relative to its origin. Options for this are (+) for a new/additional feature, (-) for a deselected/removed feature and (*) for a modified feature. When changing a feature, a new value must always be specified. Although bug fixes are not (always) directly function-related, they are represented here in a very similar way. A f is used to indicate a bug fix. Since some features shall be used in multiple variants, the patch node provides an option to transfer individual changes between different product variants. The black-marked input includes the entire history, while the second input (dashed line) only refers to the modifications of the last node and does not take its history into account. For the

sake of clarity, a shared feature is additionally listed with an & sign in the description. The numbers above the arrows refer to the corresponding change scenarios.

The following information, among others, can be taken from the representation:

- Although the bug fix is eventually propagated into all product variants, this happens not at the same time. While the Basic and Trendline products get the bug fix in V1.01, it takes until V1.02 for the Pro variant to get the updated calculation as well.
- The Trendline V1.01 was not released until both, the bug fix and the new feature OzoneSensor were implemented.
- With V2.00 the Basic variant changes its casing size to MEDIUM to fit all the necessary additional equipment.
- While the release of new product variants is generally not (and does not have to be) synchronized, sometimes multiple products receive an update at the same time. This can be the case, for example, when presenting products at exhibitions.

The visualization of such a product line evolution offers a quick and reliable overview of all existing functionalities. The relationships, dependencies and common features remain transparently visible. Since only changes are described, the presentation remains relatively lean and clear. Due to the lack of depth and the focus on functionality rather than implementation, it is not primarily intended for developers, but for product managers or release planning. Although features occur in several variants and are presented as shared, the implementation in individual products may vary due to technological requirements or limitations.

Naturally, this presentation also reaches its limits with an increasing number of releases and loses its main benefit - clarity. From time to time consolidation is therefore advisable. For this purpose, the current status of a variant is summarized along with all current features and forms the basis for a new diagram. This can happen, for example, at fixed times (e.g. annually) or when changing to a new major version.

If the reuse of features has been realized e.g. by (managed) cloning, the illustration of chronological evolution and features history becomes particularly important and helpful to maintain a general overview.

4.2 VARIABILITY REALIZATION WITH EA

The BCON system model introduced in [Zur14] serves as the basis for the implementation. However, it had to be adapted and changed considerably, since it was not fully SysML-compliant. For time reasons the new EA model is not yet complete. The initial focus was on the views most relevant to demonstrate the following examples.

This section attempts to provide a holistic view of changes to the model. Some of the evolution scenarios introduced before are explained in more detail and will be examined with regard to their drivers. Subsequently, the necessary changes to the model are explained and demonstrated. Finally, the results will be discussed. Although the scenarios were deliberately chosen to show different types of variability and to demonstrate a variety of drivers, they are still very close to a real product evolution.

SCENARIO (2)

Description. Due to several customer requests, the company BCON has decided to add a new feature to the *WeatherStationConnectPro*. Wind data shall be used to give a statement about the real outside temperature. Since the *WeatherStationConnectPro* already has a wind sensor, no mechanical/electrical changes are necessary. The function can be realized in software.

For the calculation of the wind chill index the following formulas shall be used[1]:

$$T_{wc} = 13.12 + 0.6215 \cdot T_o - 11.37 \cdot v_w^{+0.16} + 0.3965 \cdot T_o \cdot v_w^{+0.16} \tag{4.1}$$

$$T_{wc} = T_o + [\frac{(-1.59 + 0.1345 \cdot T_o)}{5}] \cdot v_w \tag{4.2}$$

with T_{wc} being the wind chill index (adjusted outdoor temperature), based on the Celsius temperature scale; T_o the measured outside air temperature in degrees Celsius; and v_w the wind speed at 10 m (33 ft) standard anemometer height, in kilometers per hour. Formula 4.1 is used for temperatures at or below 0 °C and wind speeds above 5 km/h. Formula 4.2 is used when the temperature is at or below 0 °C and wind speed is more than 0 km/h, but lower than 5 km/h.

Impact. This scenario results in two fundamental changes. On one hand, the new functionality (calculation of wind chill index) must be implemented, on the other hand, the interface to the display must be extended by an additional parameter (real temperature). Table 4.3 below shows a summary of these changes as well as their corresponding drivers and the example diagrams chosen to illustrate them. The representation of this table will be used for the other scenarios as well, because it allows a quick and easy overview and comparison. The evolution steps refer to the basic feature evolution steps listed in Table 3.1, the drivers were introduced in Section 3.1 and the variability types were discussed in Section 2.2.1.

Scenario	(2) *WeatherStationConnectPro* shall add the influence of wind to calculate real outdoor temperature.	
Change(s)	1. add new functionality	2. modify display interface
Evolution Step(s)	ES1 - optional feature creation	ES6, ES8 - alternative (coexisting) feature addition
Driver(s)	Effect Chain	Information Flow
Variability Type(s)	Option	Alternative (multiple)
Example Diagram(s)	parametric diagram	block (interface)

Table 4.3: Summery Scenario (2)

1 Climate Normals, Government of Canada, accessed 7 July 2020, https://www.canada.ca/en/environment-climate-change/services/climate-change/canadian-centre-climate-services/display-download/technical-documentation-climate-normals.html

Realization. The required calculation for the wind chill index can be represented in a parametric diagram (Figure 4.2). The necessary signals (temperature and wind speed) are already provided here. The diagram also contains the analog-digital conversion of the sensors. Assuming that the wind speed is provided in meters per second, an additional conversion is required. The additional output signal (T_{wc}) must be defined accordingly. To simplify matters, only one formula (4.1) is shown. The additional (optional) elements are outlined in red.

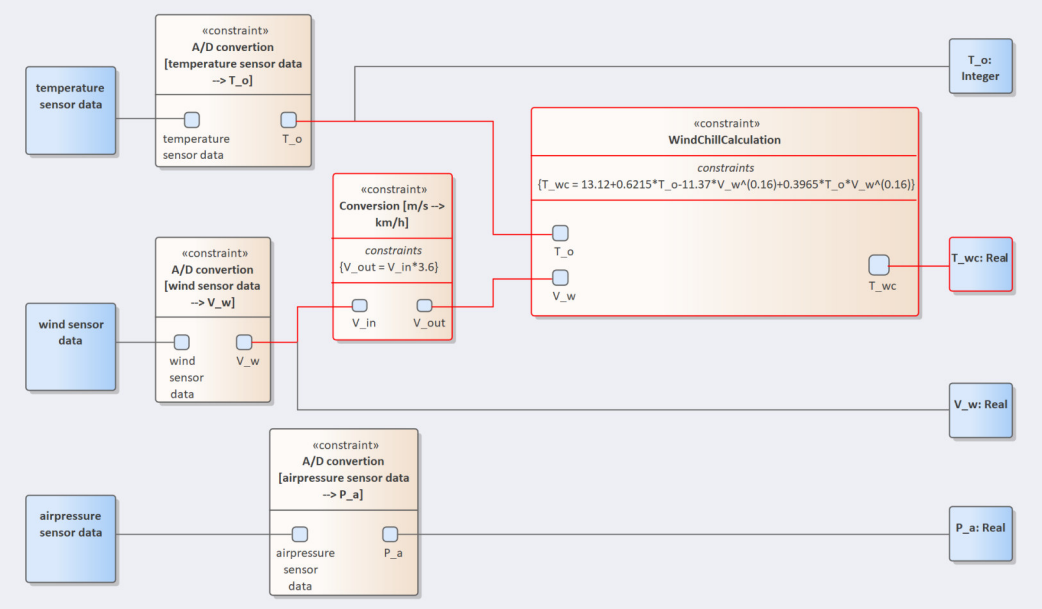


Figure 4.2: Modified parametric diagram

For the extension of the display interface, there are several ways to model the desired result. Only a single block can be defined for the interface, representing which signals the display can access. In that case everything is described in it and internal constraints can be assigned to each property (signal), keeping the model as lean and clear as possible. Figure 4.3 shows another possibility. The display interface is specialized by its three product variants. Common properties can be inherited automatically. The selection mechanism can choose one of the specializations. An advantage is that the model makes it easier to see which properties the respective variant have. A disadvantage is that if only two variants share a property, it must be defined twice (e.g. P_a).

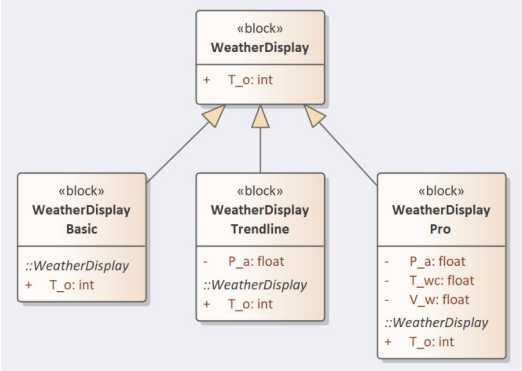


Figure 4.3: Altered display interface

Discussion. Both changes can be implemented with relatively little effort. Additional (optional) blocks, as well as additional properties can be equipped with variation points. The selection (also multiple) can be realized using pure::variants (p::v).

SCENARIO (4)

Description. Product management at BCON has decided to integrate a new feature into the two products *WeatherStationConnectTrendline* and *WeatherStationConnectPro*. A new sensor shall be able to measure the ozone values and give the user a recommendation for outdoor sports. For this purpose, the already existing interface used for several sensors shall be extended.

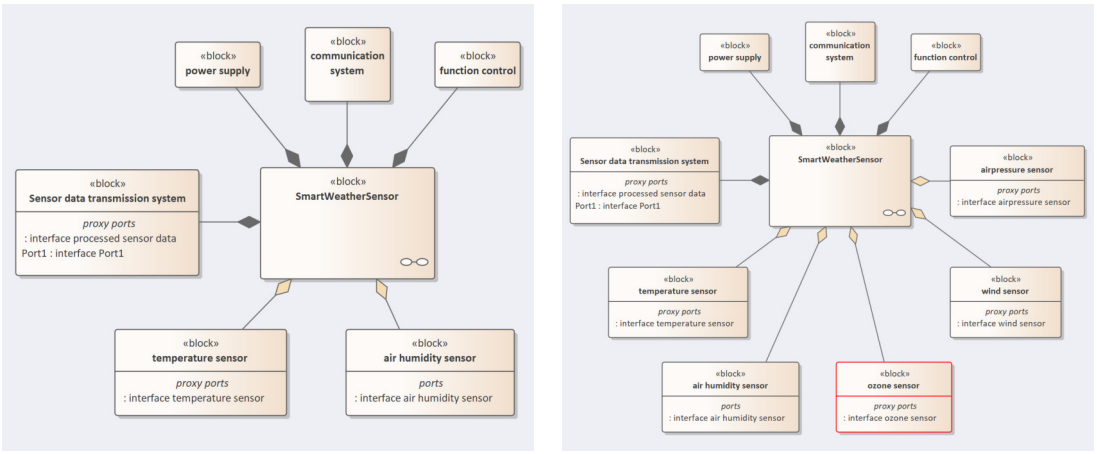
Impact. Since the product line is already designed for different sensors, the integration of a new sensor is initially straightforward. A new sensor must be introduced into the system context and the corresponding interface has to be extended. However, the higher the level of detail in the model, the more potential points of variation can arise even from this supposedly simple modification. For example, does the physical interface differ, are additional hardware components (transceivers) necessary for communication, or are there variations in data rate or format? Not all aspects can be dealt with in this example, so in the following we assume that the lowest point of variation is at the sensor interface and that all characteristics are already well defined with this selection (cf. 2.2.2).

Scenario	(4) *WeatherStationConnectTrendline* and *WeatherStationConnectPro* shall have an ozone sensor to give advice for outdoor sports.
Change(s)	1. add new component (sensor)
Evolution Step(s)	ES6, ES8 - alternative (coexisting) feature addition
Driver(s)	System Context
Variability Type(s)	Alternative (multiple), Topological
Example Diagram(s)	block (system context, interface)

Table 4.4: Summery Scenario (4)

Realization. To expand the sensor interface, two aspects must be considered. First, the sensor interface definition must be extended. All sensors are part of the SmartWeatherSensor system. Figure 4.4 shows the necessary extension. While Figure 4.4a shows the SmartWeatherSensor system with only two sensors, Figure 4.4b shows 5 sensors in total. The new ozone sensor is outlined in red. This aspect specifies the affiliation and structural arrangement of the sensors as well as all other components connected to the SmartWeatherSensor system.

The second aspect concerns the interface or the description of the data transferred via it. This is represented by a port definition and an interface block for each sensor. In case of an additional sensor, this description must be extended accordingly. Figure 4.5 shows the necessary changes and the new elements for the ozone sensor.



(a) initial sensor interface definition (b) additional sensors

Figure 4.4: Sensor interface definition

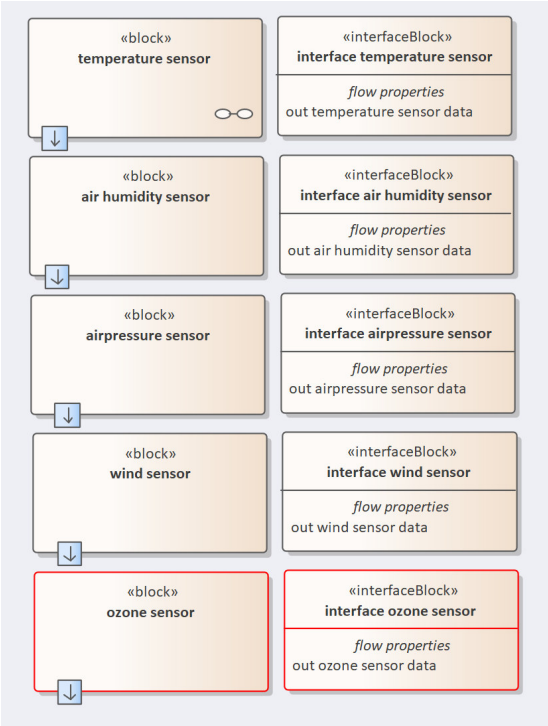


Figure 4.5: Sensor interface description

Discussion. By extending the interface with additional sensors, "only" further optional/alternative elements must be added. This can be realized relatively easily with EA and by means of p::v the sensors can be assigned to the corresponding product variants afterwards. The procedure is quite straightforward. However, the problem of open variability shall be demonstrated with this example. Assuming that the interface is repeatedly extended, so that ultimately 18 sensors must be supported. An exemplary representation is shown in Figures 4.6 and 4.7. Both the interface definition and its description gain in complexity and an overview is nearly impossible.

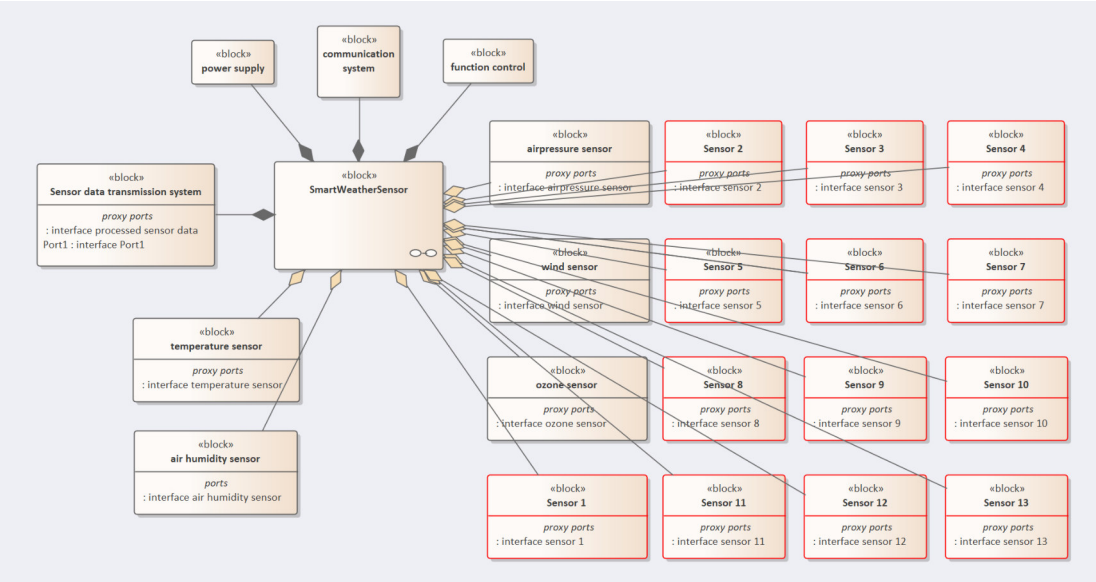


Figure 4.6: Multi sensor interface definition

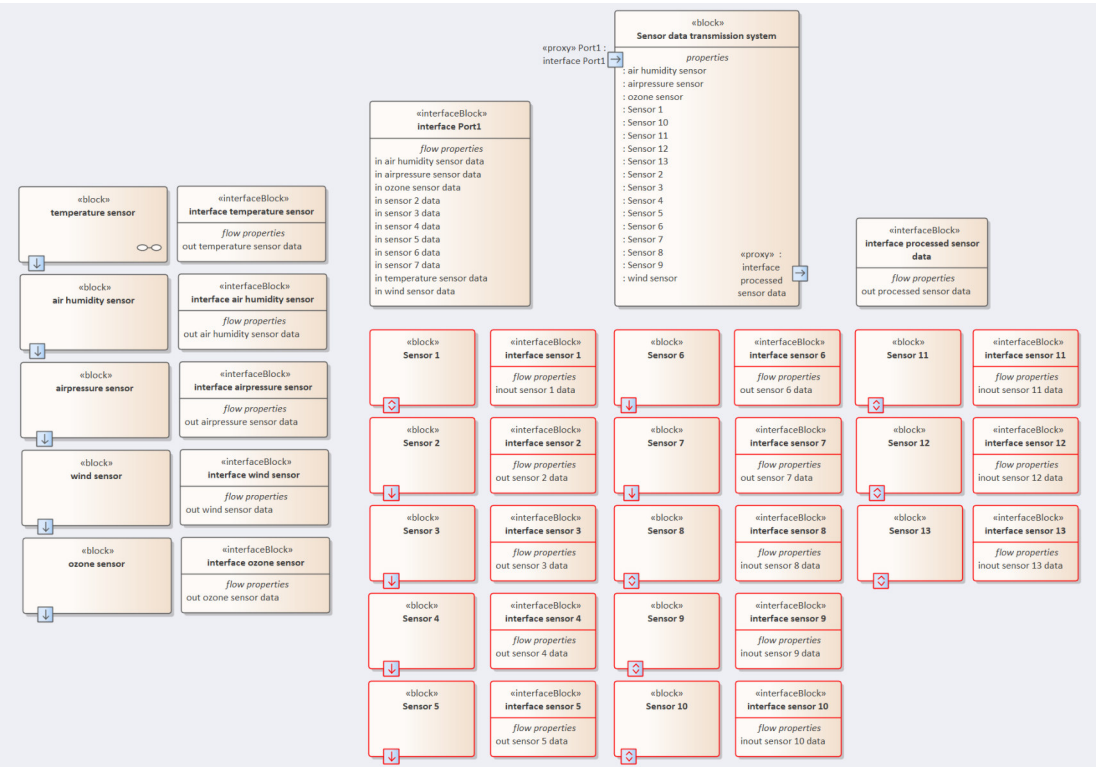


Figure 4.7: Multi sensor interface description

SCENARIO (6)

Description. Intensive tests have shown that a rounding error in the temperature calculation can cause an incorrect result. Since this calculation is used in all products, it affects them all.

Impact. Bug fixes and maintenance are not included in the feature evolution steps in Table 3.1 because they are not considered planned activities. Although maintenance in the sense of continuous improvement and refactoring may well be considered plannable, both activities are rather meant to be applied ad-hoc. However, it is often precisely for these reasons that adjustments must be made to the model. In the list of variant drivers, they have therefore been listed as a special case and this scenario has been deliberately included as an example. Depending on the type and extent of the defect, all aspects of system modeling can be potentially affected. In this example, the calculation is defined as a block (containing the formula) in a parametric diagram and can therefore be easily corrected without affecting other elements. A presentation of the realization was omitted because it does not yield any gain in knowledge.

Scenario	(6) The temperature measurement calculation is wrong in all products.
Change(s)	1. implement bug fix
Evolution Step(s)	—
Driver(s)	Bug fix / Maintenance
Variability Type(s)	—
Example Diagram(s)	—

Table 4.5: Summery Scenario (6)

Discussion. In an over-exposed (150%) model, a later correction of defects is usually less time consuming and error-prone compared to other methods. Since all of the content is in a shared model, defects can be corrected both in the common part and in the special parts. As soon as the variants are newly generated, the changes are automatically applied to all of them. However, it depends on the nature of the defect. If only certain variants are affected or if the correction has to be implemented differently in variants, the introduction of additional variation points must be considered. The more variation points and variants already exist, the more difficult it becomes to keep track of such dependencies and impacts.

With higher modularization (e.g. libraries), it depends on the location of the defect and the form in which the content of the module can be changed. Defects due to incorrect use of libraries (incorrect configuration, parameterization) can be more easily corrected. If a module itself or its interface needs to be changed, the effects on other users must be considered as well. It is possible that the "incorrect" behavior corresponds exactly to the expectations of another variant using the library.

The situation is fundamentally different if cloning was used to create variants. Since the clones may have evolved independently of each other, it is either difficult or impossible to determine which variants are affected by the bug or whether a variant can integrate the bug fix or not. Especially in this context, a consistent variant management e.g. through managed cloning pays off.

SCENARIO (20)

Description. For some years now, the sales figures of the *WeatherStationConnectBasic* have been declining. During a workshop, solution proposals were developed and a new potential application area was identified. For this, it is necessary that the weather station is functional at outdoor temperatures down to -50 degrees Celsius. The R&D department was able to design a concept and suggested the integration of a small additional heating element.

Impact. First, a new stakeholder need is introduced. From this, new functional requirements for the system can be derived. The introduction of new components (heating) results in new participants in the system context as well as changed behavior and communication. The impact of this change is enormous and resembles the necessary steps of a new development. Nevertheless, a large part of the already existing functions can be completely reused and just extended by new components.

Scenario	(20) *WeatherStationConnectBasic* shall be able to operate at ambient temperatures of -50 degrees Celsius.	
Change(s)	1. change requirements	2. add new component (heating)
Evolution Step(s)	ES3 - alternative feature creation	ES1 - optional feature creation
Driver(s)	System Context, Element Properties	System Context, Behavior
Variability Type(s)	Alternative (single)	Option, Topological
Example Diagram(s)	requirements diagram	state machine diagram, sequence diagram

Table 4.6: Summery Scenario (20)

Realization. The implementation is divided into several steps. First, the new requirement is defined as an alternative to the existing requirement for the operating temperature. Since other variants use the same requirements catalog, they cannot simply be changed. The new requirement results in two new (technical) system requirements.

Figure 4.8 shows the changed requirement diagram. SN-3 thus introduces a new variation point and SN-3.2 is newly added as a (mutually exclusive) alternative to SN-3.1.

Although, of course, structural changes are also necessary when new components are added in the system context, etc., these are not shown because the steps for this have already been illustrated in Scenario 4. The focus is therefore more on the changes in behavioral modeling.

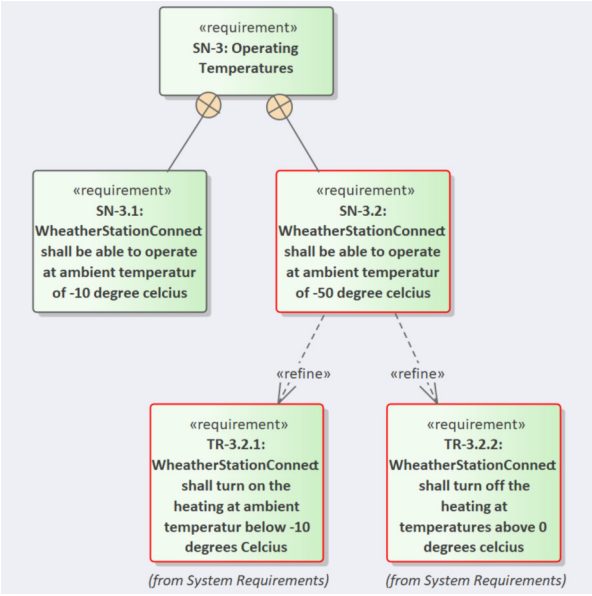
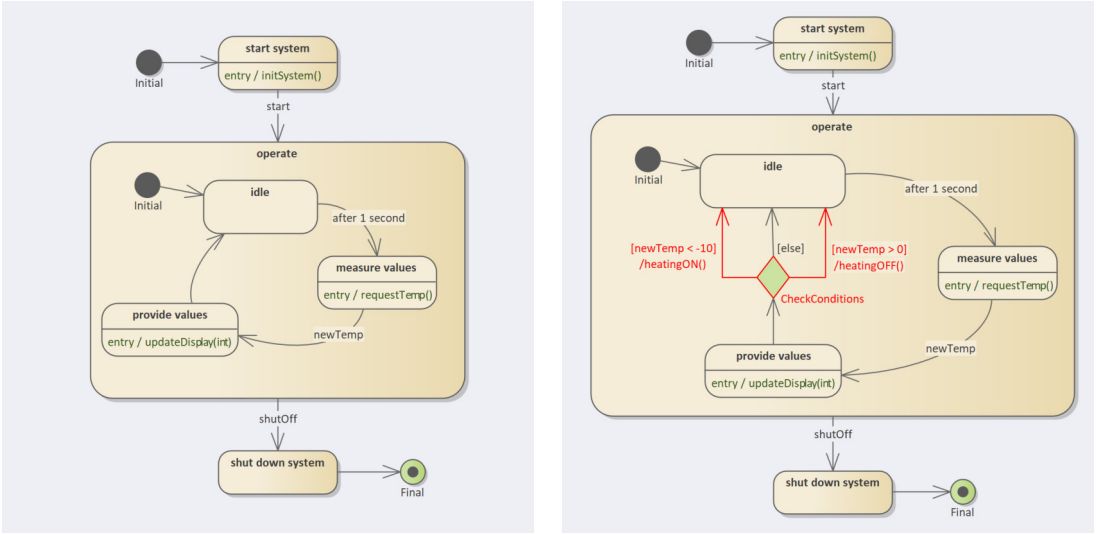


Figure 4.8: Requirements diagram

The changes in behavior are shown by means of an exemplary state machine and a corresponding sequence diagram. Figure 4.9 shows two state machine diagrams and highlights the changes required due to the additional heating in red. The original procedure (without heating) is shown in Figure 4.9a and is as follows: After the system initialization, a new measurement value is requested from the temperature sensor once per second. As soon as this temperature is known, the new value is transmitted to the display for visualization. Until the system is shut off, nothing changes in this cycle (continuous loop). Figure 4.9b illustrates the added check of the temperature condition. A decision node and two new transitions are required to implement this comparison. The initial procedure does not change at all, only two optional actions are triggered (depending on the temperature value), that switch the heating ON or OFF.



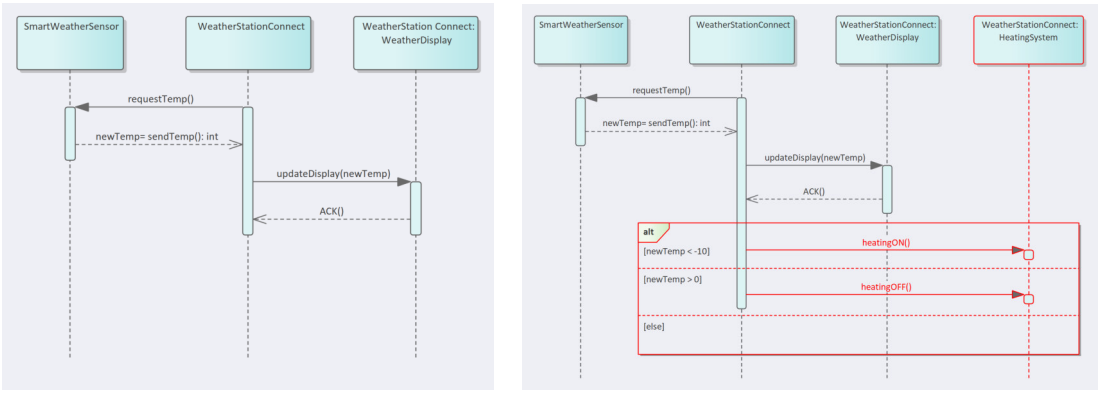
(a) initial state machine

(b) modified state machine

Figure 4.9: State machine diagram

To illustrate the change in communication, Figure 4.10 shows the sequence diagram corresponding to the above state machine. Figure 4.10a shows the initial communication between the sensor and display only. In order to keep the diagram easy to read, the communication is simplified.

With the additional heating, a new object (HeatingSystem) is introduced in Figure 4.10b. This object adds its own lifeline to interact with the other components. Since the heating is optional, its communication is shown in a new *alt* fragment. This fragment allows to define conditions and ultimately model different messages depending on their evaluation result. Thus the dependence of the transmitted message on the temperature value can be depicted directly. The messages *heatingON()* and *heatingOFF()* are intended to control the heating, this can be done by activating a relay, for example. This is why no response is shown for either message. If the heating is not included in a variant, both the object (HeatingSystem) and the fragment can be hidden using constraints, which again leads to Figure 4.10a.



(a) initial sequence diagram

(b) modified sequence diagram

Figure 4.10: Sequence diagram

Discussion. Adding new requirements as elements and assigning constraints to them is not a problem. Changing their parameter values, however, is. As shown in the example, it is necessary to insert a new requirement, even if only one value changes. No variation point can be set to the parameters (Operating Temperature) or their values, because the requirements are considered only as text. With an increasing number of different parameter values, this can lead to greater confusion in the model. Especially when variants often differ in their non-functional properties. For example, because different performance or security is required.

The behavior diagrams already offer intrinsically the possibility to model dependencies and optional/alternative elements through decision nodes, fragments, etc. This allows variability to be represented quite well with a mixture of Annotation and Conditional Execution – the actual behavior is thus decided at run/view time. Nevertheless, there is a problem with changing the decision criteria. The values at decision nodes, fragments and transitions cannot be constrained and are only represented as text. This means that a change in the criteria automatically leads to a new transition/message (including further annotations). As the number increases, the diagram will quickly become overloaded. Figure 4.11 shows an example of just two more transitions and the two annotations (selection criteria) for their corresponding variants *heatEarly* and *heatLate*. The *heatLate* variant differs only in the threshold values at which the heating is switched ON and OFF. Yet, the diagram looks much more complex than Fig. 4.9b. Transitions only allow external constraints, which is not beneficial for clarity.

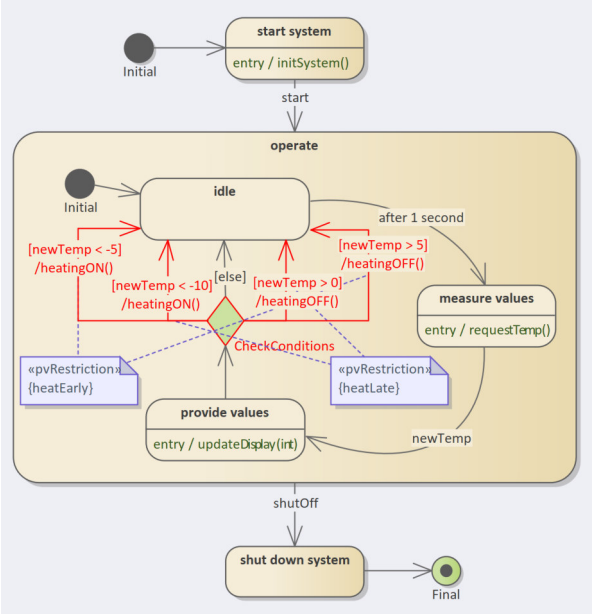


Figure 4.11: State machine with multi transition

SCENARIO (21)

Description. Due to its increasing popularity and decreasing price, the *WeatherStationConnectTrendline* shall also support digital temperature sensors. For this purpose the sensor interface must be extended accordingly.

Impact. This scenario is very similar to Scenario 4, but differs in one aspect. By changing the signal type that is transmitted via the interface from analog to digital, the downstream processing also changes. Functions such as A/D conversion or the time averaging of measured data are no longer necessary, as these are already handled by the digital sensors themselves. Thus the allocation of these functions changes.

Scenario	(21) *WeatherStationConnectTrendline* shall support digital temperature sensors.	
Change(s)	1. add new component (sensor)	2. modify signal processing
Evolution Step(s)	ES6, ES8 - alternative (coexisting) feature addition	ES3 - alternative feature creation
Driver(s)	System Context, Information Flow	Effect Chain, Allocation
Variability Type(s)	Alternative (multiple), Topological	Alternative (single), Topological
Example Diagram(s)	internal block (sensor)	parametric diagram

Table 4.7: Summery Scenario (21)

Realization. The extension of the sensor interface was already shown in Scenario 4 and is therefore not illustrated here. The focus is on the shift of responsibilities with regard to the analog-to-digital conversion. Figure 4.12 therefore shows the internal block representation of the new sensor, that contains the necessary conversion. It is important to note that this sensor must be either additionally modeled (as in this case) or must provide both signals (analog and digital), since both analog and digital sensors can be used in a variant. Possibly even in any combination.

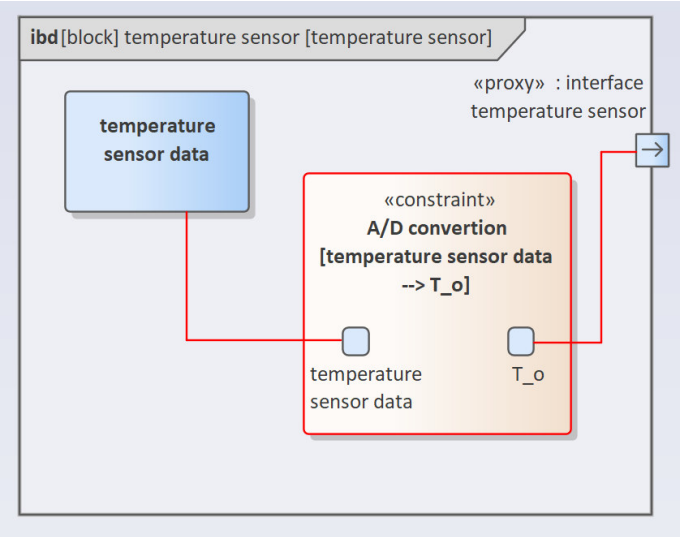


Figure 4.12: Internal block diagram of digital sensor

Figure 4.13 is a detailed extract from Figure 4.2 and shows in particular the extension of the parametric diagram by the new digital input for the temperature data. If a digital sensor is used, both the analog input and the A/D conversion are not needed and are therefore grayed out. However, for analog temperature sensors, both are still available and can still be used. The data type for the new digital input is directly defined as integer and equals T_o. It can therefore be routed directly to the output, but may also be used for the *WindChillCalculation*.

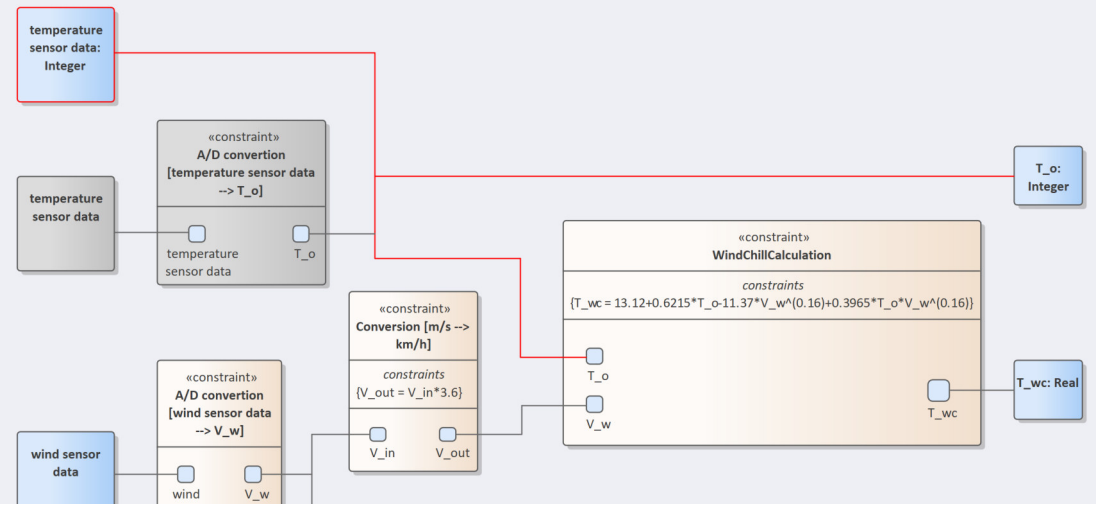


Figure 4.13: Parametric diagram with new digital sensor

Discussion. The relocation of functionalities is usually not a major problem. Blocks can always be attached with constraints. However, the larger distribution of these constraints (across several diagrams) increases topological dependencies. Potentially, functions could also be modeled redundantly. In this example it is assumed that the two A/D conversions are not identical (e.g. different resolution). However, they could also be identical components, which then have to be modeled several times at different locations. As far as the signal flow is concerned, special attention must be paid to the distribution. In this example both signals were simply combined and considered to be equivalent. If a combination of analog and digital sensors exists in one variant, it must be decided via annotations which signal is used for the calculation and for displaying. Perhaps the physical location of the sensors is relevant for this decision.

SCENARIO (22)

Description. After introducing digital temperature sensors for the *WeatherStation-ConnectTrendline*, BCON has decided to increase the resolution of these sensors. In order to realize the higher resolution, the data type must be changed from integers to decimal numbers.

Impact. This time the change no longer only affects the interface description (new interface) or the signal transmission (changed signal path, physical description), but also affects the content of the data to be transmitted itself. As a result, variation points occur wherever this data is processed. Since no variant with exclusive digital or analog

sensors is intended, the implementation is not an option but a new (coexisting) alternative for the sensor interface. However, for the variation points on the components that ultimately process the data, there is only one additional (mutually exclusive) alternative – the data arrives either as integer or float/real and must be processed accordingly.

Scenario	(22) *WeatherStationConnectTrendline* shall support a digital high precision temperature sensor.
Change(s)	1. modify data processing
Evolution Step(s)	ES_3 - alternative feature creation
Driver(s)	Information Flow
Variability Type(s)	Alternative (single), Topological
Example Diagram(s)	—

Table 4.8: Summery Scenario (22)

Discussion. No constraints and therefore no variation points can be attached to parameters respectively their values or data types. It is therefore not possible to realize a variation of the data type automatically. The only solution would be to create a second (alternative) signal with a different data type and to insert an additional variation point at each position where this signal is used. Depending on how often the signal is used, each variation introduces significant topological dependencies. Regardless of how the variation is implemented, each use of the new signal must be individually assessed and verified. In particular, a change in data type can cause problems such as rounding/calculation errors, display errors, transmission problems. Other components may require a special format and are not flexible in this respect – common issues arise in relation to the representation of date/time.

Although changes to names, data types or parameters have proven to be very difficult and time-consuming, it should be noted that SysML itself follows the Single Source of Truth (SSOT) principle and is therefore not the primary cause of this problem. SSOT focuses on the handling of intentional redundancy and aims to provide correct and reliable data at all times. Parameters and properties that are used multiple times should therefore only be defined once and distributed to the appropriate places in the model using references. When working with EA, however, several problems arose, so that it was not always possible to work reliably with references to existing properties and parameters. The resulting repeatedly defined properties led to ambiguities and had to be checked and adjusted individually when changes were made. Especially data types could not be assigned and managed centrally.

EXCURSUS

As already mentioned, the combination of Enterprise Architect (EA) and pure::variants (p::v), which was also used for the feasibility study, has been examined in more detail. In particular with regard to the possibilities of representation and granularity of the variation points. The insights gained in this respect will be briefly explained in the following. Since the combination of these tools is not unusual in practice, the results may help to develop a better understanding of what is possible and where difficulties may arise. The aim is not to undermine their use, but only to raise awareness of possible pitfalls and limitations.

In order to examine the possibilities and potential problems more closely, a very simple EA model was first generated. Since EA stores the model content in a database using the Microsoft Jet Database Engine, it can be easily read and processed by other programs as well. With the help of Microsoft PowerShell a script (cf. A.1) was created to extract the Extensible Markup Language (XML) structures stored in all relevant database tables and display the content as readable strings. This data could then be transferred to a Version Control System (VCS), making it possible to ultimately track and evaluate all changes to the model. Once this processing chain was set up, small changes were made to the model and committed to the VCS incrementally. The impact on the database content could be evaluated immediately with the internal diff tool.

Figures 4.14 & 4.15 show an example of such a database change for an added constraint element. The commit dialog depicted in Figure 4.14 shows, that 8 lines have been added. A comparison with the base version, shown in Figure 4.15, reveals the specific entries and shows in which table the changes were made. With this procedure, a good insight could be gained into how EA stores and structures the elements internally in its database. One by one, different diagrams were created and filled with additional elements.

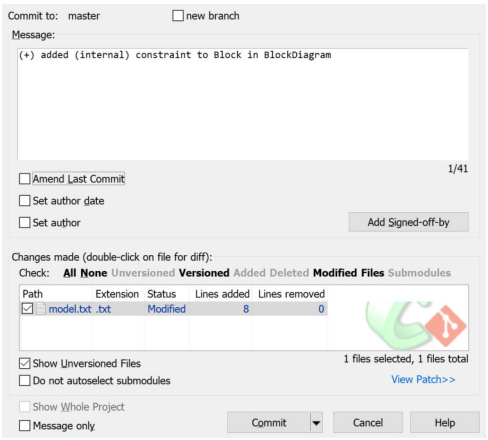


Figure 4.14: Commit dialog

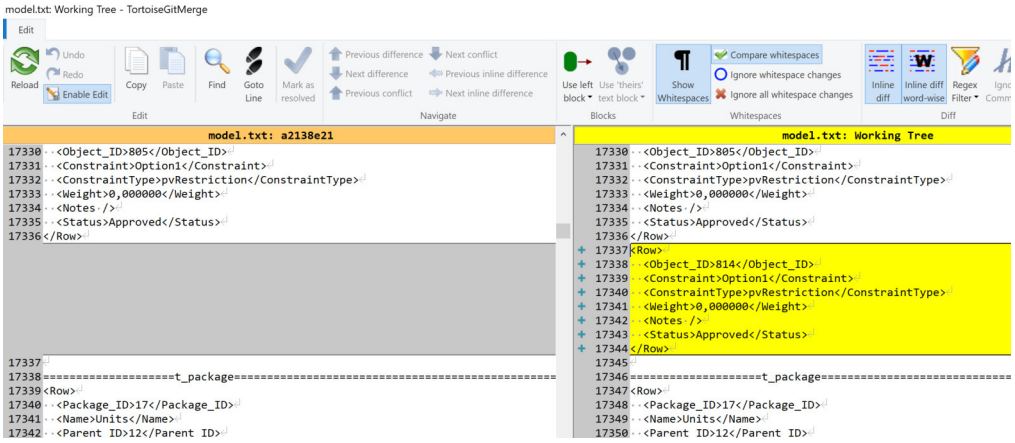


Figure 4.15: Differences (highlighted) of DB content compared to base version

The following list shows the main sequence of changes made:

(1) Create new diagram
(2) Add new elements
(3) Add new connections (relation between elements)
(4) Add (internal) constraints to elements
(5) Add (external) constraints to elements/connections
(6) Add attributes and methods to elements
(7) Add parameter and (default) values
(8) Change values/data types

Since the exchange mechanism between EA and p::v is based on constraint elements, special attention was paid to them. The objective was to identify to which elements a constraint can be (internally) added and to which elements/connections external constraints can be attached. Table 4.9 shows a summary of the findings. It indicates in which database table the respective types are stored. If it was possible to define internal constraints, the table in which they are stored is also indicated. An attempt has been made to give as general a statement as possible across the various types of diagrams. Unfortunately, this was not always possible, especially for the connections. In all cases though, it was possible to define either an internal or external constraint for elements and connections.

TYPE	TABLE(S)	INTERNAL CONSTRAINT	CONSTRAINT TABLE(S)	EXTERNAL CONSTRAINT
Element	t_object t_diagramobjects	partly	t_objectconstraint	yes
Connector	t_connector t_diagramlinks	partly	t_connectorconstraint	partly
Attribute	t_attribute	yes	t_attributeconstraints	no
Method	t_operation	yes	t_operationspres	no
Parameter	t_attribute t_operationparams	no	-	no
Values (Defaults)	t_attribute	no	-	no
Data type	t_attribute t_operation t_operationparams	no	-	no

Table 4.9: EA internal/external constraint options

Methods and attributes of elements are not accessible for external constraints, but allow to define a variation point internally. However, as Table 4.9 shows, there is a problem at parameter level. No variation point can be assigned to their number, name, type and value. This becomes even more obvious considering that these properties are not stored in the database as separate objects, but as text properties of their parent elements. Similar restrictions were also shown in the case study [Tru+10]. Their list of supported SysML elements also lacks parameters, values and data types. As mentioned before, this limitation entails some disadvantages. For example, Etxeberria et al. explicitly states the importance for the realization of varying (non-functional) requirements, which are usually expressed by parameters [ESB10].

4.3 EVALUATION

In Table 4.10 below, the mechanisms identified in Section 3.4 are evaluated according to the criteria described in Section 3.5. Most of the information presented is based on the experience gained during the feasibility study. Since not all mechanisms were used, information had to be supplemented in part by other sources (cf. [Pat11][ZDB16]). Aspect-/Delta-Orientation was omitted due to its lack of practical relevance.

MECHANISM	METHOD(1)	EFFORT(2)	GRANULARITY	ISOLATION	OPEN	EXPLICIT	INDEPENDENCE	DEFAULTS
Cloning	all	low →	any	yes	no	no	high	no
Libraries	90% 150%	low(3) ↗	modules packages	yes	yes	yes	medium	no
Module Replacement	90% 150%	medium(3) ↘	modules packages	yes	yes	no	medium	no
Conditional Compilation	150%	high ↘	any(4)	no	no	yes(5)	low	yes
Conditional Execution	all	low(6) ↗	behavior	no	no	no	low	no
Polymorphism	90% 150%	medium ↘	modules packages	yes	yes	yes	medium	no

Table 4.10: Evaluation of Variability Realization Mechanisms

Remarks:

(1) The method refers to the variation management approaches presented in Section 3.2: Managed Cloning, Product Line (90%), Production Line (150%).

(2) Effort describes the initial effort first, but also tries to make a statement whether the continuous effort is constant (→), increasing (↗) or decreasing (↘).

(3) Both approaches are essentially very similar. However, Module Replacement is considered more flexible and customizable, so the effort decreases over time, while Libraries are considered as prefabricated assets and as such are difficult to customize and may need to be replaced to support new functionalities.

(4) In principle, the granularity is not restricted, but no variation points at parameter level were possible in the tool combination used.

(5) Variation points are only visible and thus explicit in the base model (150%), not in the the derived variant models (100%).

(6) The initial effort required to introduce decision points is small, but maintaining dependencies (across multiple diagrams) quickly adds complexity and effort. In particular because of their low explicity.

IMPACT OF VARIANT DRIVER ON REALIZATION

The following Table 4.11 shows a juxtaposition of the variant drivers introduced in Section 3.1 and the 8 SysML diagram types. The representation is intended to provide an overview of which drivers have low, medium or high influence on which views/diagrams.

VARIANT DRIVER	ACTIVITY	BLOCK	PACKAGE	PARAMETRIC	REQUIREMENT	SEQUENCE	STATE	USE CASE
System Context	◐	●	●	◐	●	◐	◐	◐
Effect Chain	◐	◐	○	●	○	○	◐	○
Information Flow	◐	●	●	◐	○	◐	○	○
Behavior	●	○	○	○	◐	●	●	●
Element Properties	◐	●	●	○	●	◐	◐	○
Allocation	○	●	●	◐	○	○	○	○
Composition	○	●	●	◐	○	◐	○	◐
Bug fix / Maintenance	●	●	●	●	●	●	●	●

● = high impact ◐ = medium impact ○ = low impact

Table 4.11: Effects of Variant Drivers on SysML diagrams

The information contained in Table 4.11 is intended to serve as a basis for discussion and to provide an exemplary classification or rough orientation. The actual impact of the drivers on the views must be analyzed within the individual modeling and variant management approaches. With the help of experts and practitioners, these data could be discussed and further elaborated, e.g. through interviews or surveys. The individual influences can vary greatly. The presentation should therefore serve as a stimulus to identify own driving factors and to record the frequencies of their change. If, for example, it is known in advance which variant drivers are usually to be expected, it can be deduced which views/diagrams will be affected more frequently by changes. Modularization and changeability can thus be taken into account early on in the implementation of the model. As a result, the system model is much better prepared for change.

IMPACT OF VARIABILITY TYPE ON REALIZATION

As already suspected in Section 2.2.1, the examples of the feasibility study have demonstrated that open and topological variability is particularly difficult to cope with.

Open Variability. It is difficult to prepare for open variability and frequent changes or extensions require continuous effort. The only countermeasure is the awareness of this problem and the preparation of the model for frequent changes and easy extensibility at these points, e.g. by modularization (polymorphism, libraries, plugins). Explicit modeling and adherence to variation points helps to achieve this. Consider open/closed principle.

Topological Variability. Topological variability is even more difficult to manage. A great system understanding is necessary to keep the effects of such far-reaching changes completely transparent. Possibilities to counteract this are the sparing use of variation points and the division of a complex system into several components (divide and conquer), each of which implements variability in a self-contained manner and thus follows the basic principle of maintainable systems (low coupling, high cohesion). Especially important is the clear separation of common and variable parts.

IMPACT OF GRANULARITY ON REALIZATION

If a variation point with the required granularity is not possible, it must be handled on the next higher level of granularity. This may lead to redundancy if, for example, several methods have to be created that differ only in their return value. The use of an appropriate mechanism that supports any granularity can be beneficial.

RECOMMENDATIONS

The following Table 4.12 summarizes once again the main advantages and limitations of the identified variability realization mechanisms, and provides a short recommendation for their use.

VARIABILITY MECHANISM	PRACTICAL BENEFITS	PRACTICAL CHALLENGES	RECOMMENDATIONS
Cloning	Quick introduction with low effort. Independence from other variants. Applies to any artifact type and size.	High long-term maintenance effort. Incomplete propagation of changes and bug fixes, leading to lower model quality.	Quick removal of short-lived clones. Periodic clone evaluation and refactoring to other mechanisms in case of maintenance problems or increasing number of variants.
Libraries	Shared model content is packaged into model libraries (separation and division).	It can be difficult to reproduce errors, if interna of a library are not visible. Search and identification may be difficult.	Well suited for variants whose contents should not be changed and kept consistent.
Module Replacement	Separation of common and variant elements. High flexibility. Each variant element can evolve in isolation.	Hard to identify variation points and variant elements.	Managing variation points and variant modules with appropriate tool support.
Conditional Compilation	Rather easy to introduce. No limited granularity. Explicit variation points. No language support necessary. No model evaluation penalties.	Annotated models are harder to understand due to variation point nesting, tangling, scattering, etc. They have the tendency to erode and are harder to maintain during evolution.	Analyze and manage the complexity of annotated variation points against variability erosion. Refactoring in case of maintenance problems.
Conditional Execution	Easy to introduce and understand, as modeling experts already know the language features. Variability impact can be analyzed with model analysis tools.	Hard to distinguish between the variation logic and common model content. Variation logic adds some evaluation penalties in the model.	Separate the variability concern clearly from the common model content. Follow naming rules for the variabilities. Consider using Conditional Compilation instead.
Polymorphism	Separation of common and variant elements. High flexibility. Each variant element can evolve in isolation.	Lower efficiency due to fragmentation of variant elements. Increased risk of defects.	Avoid in embedded context if runtime variability is not necessary. Consider using Module Replacement instead.

Table 4.12: Guideline and Recommendations

5

CONCLUSION

5.1 RESULTS

It could be shown that the mechanisms for the realization of variability identified by Patzke [Pat11] can in principle be transferred from the code level to the model level. They differ slightly in their application, possibilities and limitations. The criteria for their evaluation also had to be adapted and supplemented. In contrast to the code level, however, modeling requires a greater focus on a general reuse approach. On one hand, this decision limits the choice of mechanisms, but on the other hand it offers a higher potential for the reuse of model elements in the overall system context.

Furthermore it could be shown that the effort for variant modeling increases with the level of detail in the model. If, for example, a new sensor is added to the weather station, it can be easily displayed as a new component in the system context. In case the model is used to support communication between stakeholders, this representation can be sufficient. If, on the other hand, the model is used for the specification of the system, all aspects (such as communication, interfaces, data rate, parameters, behavior, etc.) must be considered. This increases the necessary effort immensely, especially if dependencies to other products (within a product line) exist or arise because of this change.

With regard to the research questions formulated at the beginning of this thesis, a brief reference to the places where the points were discussed is given below:

RQ1 A list of variability realization mechanisms suitable for the use with SysML models can be found in Section 3.4. In particular, Figure 3.7 gives an overview of all identified and examined mechanisms.

RQ2 The key performance indicators used to asses the variability mechanisms are listed and explained in Section 3.5. A practical evaluation of the identified mechanisms is shown in Table 4.10. Guidelines and recommendations for the application of these mechanisms are summarized in Table 4.12.

RQ3 Limiting factors are discussed throughout the thesis. Of particular note are the Sections 3.2 and 3.3, where management approaches as well as aspects of tool support are discussed. Both limit the choice of mechanisms. Further restrictions are also listed in Table 4.10.

Although a 150% model has major difficulties, especially in terms of parallel development, the current trend seems to be in this direction. At least that is what the few relevant studies indicate. From the practical work it can be confirmed that modeling and managing variability is most comfortable with a 150% model. It provides

a good separation between feature and implementation view, allows fast and automatic generation of variants and supports all types of variability. Nevertheless, several limitations emerged that should not be underestimated. Parallel working is difficult, which could be improved by a VCS, but not solved in principle. Further aspects are the (tool-dependent!) granularity of variation (im-)possibilities and the increasing complexity and incomprehensibility of the 150% model. This was particularly evident in cases of topological variability or with a high number of variants (open variability).

5.2 DISCUSSION

Although some guidelines, recommendations and challenges could be identified and formulated with the conducted feasibility study, they do not cover the wide range of possibilities in variant modeling sufficiently. The research carried out therefore represents only a small part of it. Even if, to the best of my knowledge and belief, some statements are transferable and generally valid, there is no proof of this. More detailed investigations are necessary to obtain a more conclusive result, especially with regard to other modeling methods and tools.

5.3 FUTURE WORK

In order to obtain a broader overview and more general statements/guidelines, further research must be conducted. Examples of further areas for this would be:

- extend the BCON model for reference and demonstration purposes
- investigate the effect of different modeling methods on variability
- investigate the influence of tool features on the realization of variability more
- what possibilities are offered by other tools (e.g. regarding adaptability [Bil+19])

A survey of practitioners and system developers could provide valuable insights into which approaches are actually used in industry today and whether the problems and difficulties described in Chapter 4 are perceived in a similar way. The list of suggested tool capabilities (cf. Table 3.2) can thereby be compared and extended if necessary. A comparison of the typical variant drivers would also be useful to check the validity of the information given in Section 3.1.

APPENDIX

```powershell
# Step 1: Open EA project file
$ModelPath = "Projekt_EA.EAP"

function Convert-XmlElementToString
{
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)]
        $xml
    )

    $sw = [System.IO.StringWriter]::new()
    $xmlSettings = [System.Xml.XmlWriterSettings]::new()
    $xmlSettings.ConformanceLevel = [System.Xml.ConformanceLevel]::Fragment
    $xmlSettings.Indent = $True
    $xw = [System.Xml.XmlWriter]::Create($sw, $xmlSettings)
    $xml.WriteTo($xw)
    $xw.Close()
    return $sw.ToString()
}

Set-Alias "Log" "Write-Host"

# Step 2: Open current project file in EA
$EA = New-Object -ComObject "EA.Repository" -Strict
$Res = $EA.OpenFile($ModelPath)
$EA.App.Visible = $True

# Step 3: Get list of DB tables
$EATables = ([xml] $EA.SQLQuery("SELECT * FROM usystables")).EADATA.Dataset_0.Data.Row

$EADB = @{}
$EATables |
? ToVer -eq "9.9.9" | # skip deprecated tables
Select -ExpandProperty TableName |
% {
    Log ". read $_"
    $EADB.$_ = ([xml] $EA.SQLQuery("SELECT * FROM $_")).EADATA.Dataset_0.Data.Row
}

# Step 4: Convert DB content into (readable) strings
$EADBDump = ""
$EADB.GetEnumerator() |
% {
    $EADBDump += (("=" * 20) + $_.Name + ("=" * 50)) + "`r`n"

    $xml = $_.Value
    if ($xml) {
        $EADBDump += Convert-XmlElementToString -xml $Xml
        $EADBDump += "`r`n" *2
    }
}

# Step 5: Show DB content
Write-Host $EADBDump
```

Listing A.1: PowerShell script to export EA model content

BIBLIOGRAPHY

[AZ13] Albert Albers and Christian Zingel. "Challenges of Model-Based Systems Engineering: A Study towards Unified Term Understanding and the State of Usage of SysML." In: *Lecture Notes in Production Engineering*. Springer Berlin Heidelberg, 2013, pp. 83–92. DOI: 10.1007/978-3-642-30817-8_9.

[Ana+13] H. Anacker et al. "Solution patterns to support the knowledge intensive design process of intelligent technical systems." In: *Proceedings of the International Conference on Engineering Design, ICED* 6 (Aug. 2013), pp. 101–112.

[ADG14] Harald Anacker, Roman Dumitrescu, and Jürgen Gausemeier. "Design Framework for the Integration of Cognitive Functions Based on Solution Patterns." In: *Gausemeier, J. and Rammig, F. and Schäfer, W. (Hrsg.): Design Methodology for Intelligent Technical Systems, Springer, Berlin*. 2014.

[Ape+13] Sven Apel et al. *Feature-Oriented Software Product Lines*. Springer, 2013.

[Bec03] Martin Becker. "Towards a General Model of Variability in Product Families." In: *Software Variability Management Workshop*. Jan. 2003, pp. 19–27.

[Bec17] Martin Becker. *Textbook: Product Line Engineering*. v2.1. 2017.

[BZ18] Martin Becker and Bo Zhang. "How do our neighbours do product line engineering?" In: *Proceeedings of the 22nd International Conference on Systems and Software Product Line*. ACM Press, 2018, pp. 190–195. DOI: 10.1145/3233027.3233045.

[Ber+13] Thorsten Berger et al. "A survey of variability modeling in industrial practice." In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. ACM Press, 2013. DOI: 10.1145/2430502.2430513.

[Ber+14] Thorsten Berger et al. "Three Cases of Feature-Based Variability Modeling in Industry." In: *Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014*. Valencia, Spain, Sept. 2014, pp. 302–319. DOI: 10.1007/978-3-319-11653-2_19.

[Beu13] Danilo Beuche. "pure::variants." In: *Systems and Software Variability Management*. Springer Berlin Heidelberg, 2013, pp. 173–182. DOI: 10.1007/978-3-642-36583-6_12.

[Bil+18] Damir Bilic et al. "Model-Based Product Line Engineering in an Industrial Automotive Context: An Exploratory Case Study." In: *Proceeedings of the 22nd International Conference on Systems and Software Product Line*. Vol. 2. ACM Press, Sept. 2018, pp. 56–63. DOI: 10.1145/3236405.3237200.

[Bil+19] Damir Bilic et al. "An Integrated Model-based Tool Chain for Managing Variability in Complex System Design." In: *Models and Evolution Workshop (ME 2019), co-located with the IEEE / ACM 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS 2019)*. Sept. 2019. DOI: 10.1109/MODELS-C.2019.00045.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. "Formalizing cardinality-based feature models and their specialization." In: *Software Process: Improvement and Practice* 10 (Jan. 2005), pp. 7–29. DOI: 10.1002/spip.213.

[DAB15] Dominik Domis, Rasmus Adler, and Martin Becker. "Integrating variability and safety analysis models using commercial UML-based tools." In: *Proceedings of the 19th International Conference on Software Product Line*. SPLC. ACM Press, July 2015. DOI: 10.1145/2791060.2791088.

[Dub+13] Y. Dubinsky et al. "An Exploratory Study of Cloning in Industrial Software Product Lines." In: *17th European Conference on Software Maintenance and Reengineering*. IEEE, Mar. 2013. DOI: 10.1109/csmr.2013.13.

[Dum14] Cosmin Dumitrescu. "CO-OVM: A Practical Approach to Systems Engineering Variability Modeling." PhD thesis. Université Panthéon-Sorbonne, Paris, June 2014.

[ERZ14] Martin Eigner, Daniil Roubanov, and Radoslav Zafirov, eds. *Modellbasierte virtuelle Produktentwicklung*. Springer Berlin Heidelberg, 2014. DOI: 10.1007/978-3-662-43816-9.

[ESB10] Leire Etxeberria, Goiuria Sagardui, and Lorea Belategi. "Quality aware Software Product Line Engineering." In: *Journal of the Brazilian Computer Society* 14.1 (2010), pp. 57–69. DOI: 10.1007/bf03192552.

[FG13] Jörg Feldhusen and Karl-Heinrich Grote, eds. *Pahl/Beitz Konstruktionslehre*. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-29569-0.

[Fin10] Arlene Fink. *Conducting Research Literature Reviews: From the Internet to Paper, 3rd Edition*. SAGE Publications, Inc., 2010.

[Gal+14] Matthias Galster et al. "Variability in Software Systems - A Systematic Literature Review." In: *Software Engineering, IEEE Transactions on* 40 (Mar. 2014), pp. 282–306. DOI: 10.1109/TSE.2013.56.

[Gam+95] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 10.5555/186897. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.

[Hei+17] Jens Heidrich et al. "Systems Engineering as an Enabler for Future Innovation." In: *Tag des Systems Engineering*. Carl Hanser Verlag GmbH & Co. KG, Nov. 2017, pp. 87–96. DOI: 10.3139/9783446455467.009.

[INC15] INCOSE. *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*. 4th ed. Wiley, 2015.

[Jéz12] Jean-Marc Jézéquel. "Model-Driven Engineering for Software Product Lines." In: *ISRN Software Engineering* 2012 (2012), pp. 1–24. DOI: 10.5402/2012/670803.

[Kai14] Lydia Kaiser. "Rahmenwerk zur Modellierung einer plausiblen Systemstruktur mechatronischer Systeme." Dissertation. Fakultät für Maschinenbau, Universität Paderborn, 2014.

[Kan+90] Kyo Kang et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, 1990.

[KG08] Cory J. Kapser and Michael W. Godfrey. ""Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software." In: *Empirical Software Engineering* 13.6 (July 2008), pp. 645–692. DOI: 10.1007/s10664-008-9076-6.

[Käs12] Christian Kästner. "Virtual Separation of Concerns: Toward Preprocessors 2.0." In: *it - Information Technology* 54.1 (Feb. 2012), pp. 42–46. DOI: 10.1524/itit.2012.0662.

[KC07] Barbara Kitchenham and Stuart Charters. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Tech. rep. Keele University and Durham University, 2007.

[Kru17] Benjamin Kruse. "A Library-Based Concept Design Approach for Multi-Disciplinary Systems in SysML." PhD thesis. ETH Zurich, 2017. DOI: 10.3929/ETHZ-B-000172379.

[Lab17] Mirko Karl-Heinz Laborenz. "Variantenmodellierung und -management in der frühen Phase der Produktentwicklung." Studienarbeit TU Kaiserslautern. Oct. 2017.

[Lie+09] Jörg Liebig et al. "RobbyDBMS." In: *Proceedings of the First International Workshop on Feature-Oriented Software Development*. ACM Press, 2009. DOI: 10.1145/1629716.1629729.

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, Aug. 2008. 464 pp. ISBN: 0132350882.

[McI69] Malcolm Douglas McIlroy. "Mass produced software components." In: *Software Engineering: Report of a conference sponsored by the NATO Science Committee*. Ed. by P. Naur and B. Randell. NATO. Garmisch, Germany, Jan. 1969, pp. 79–87.

[MP14] Andreas Metzger and Klaus Pohl. "Software product line engineering and variability management: achievements and challenges." In: *Proceedings of the Future of Software Engineering - FOSE 2014*. ACM Press, May 2014, pp. 70–84. DOI: 10.1145/2593882.2593888.

[Ols15] Dan Olsen. "Problem Space versus Solution Space." In: *The Lean Product Playbook*. John Wiley & Sons, Inc, May 2015, pp. 13–22. DOI: 10.1002/9781119154822.ch2.

[ONX08] S. K. Ong, Andrew Y. C. Nee, and Q. L. Xu. *Design Reuse in Product Development Modeling, Analysis and Optimization*. World Scientific, Nov. 2008. DOI: 10.1142/6929.

[Pat11] Thomas Burkhard Patzke. "Sustainable Evolution of Product Line Infrastructure Code." PhD thesis. TU Kaiserslautern, 2011.

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. 1st ed. Springer, 2005. DOI: 10.1007/3-540-28901-1.

[RCC13] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. "Managing cloned variants." In: *Proceedings of the 17th International Software Product Line Conference*. SPLC. ACM Press, Aug. 2013. DOI: 10.1145/2491627.2491644.

[Sch19] Andreas Schäfer. "Methode zur modellbasierten Entwicklung von multidisziplinären Systemen in kleinen Unternehmen." Diplomarbeit TU Kaiserslautern. Aug. 2019.

[Sch18] Felix Schwägerl. "Version Control and Product Lines in Model-Driven Software Engineering." PhD thesis. Universität Bayreuth, Feb. 2018.

[Sve+10] Andreas Svendsen et al. "Developing a Software Product Line for Train Control: A Case Study of CVL." In: *Software Product Lines: Going Beyond*. Springer Berlin Heidelberg, 2010, pp. 106–120. DOI: 10.1007/978-3-642-15579-6_8.

[Tis+12] Christian Tischer et al. "Developing long-term stable product line architectures." In: *Proceedings of the 16th International Software Product Line Conference*. Vol. 1. SPLC. ACM Press, Sept. 2012, pp. 86–95. DOI: 10.1145/2362536.2362551.

[TK19] Juha-Pekka Tolvanen and Steven Kelly. "How Domain-Specific Modeling Languages Address Variability in Product Line Development." In: *Proceedings of the 23rd International Systems and Software Product Line Conference*. SPLC. ACM Press, Sept. 2019, pp. 155–163. DOI: 10.1145/3336294.3336316.

[Tru+10] Salvador Trujillo et al. "Coping with Variability in Model-Based Systems Engineering: An Experience in Green Energy." In: *Modelling Foundations and Applications*. Springer Berlin Heidelberg, 2010, pp. 293–304. DOI: 10.1007/978-3-642-13595-8_23.

[Wei14] Tim Weilkiens. *Systems Engineering mit SysML/UML: Anforderungen, Analyse, Architektur*. dpunkt.verlag, 2014.

[Wei16] Tim Weilkiens. *Variant Modeling with SysML*. MBSE4U Booklet Series, 2016.

[Wie17] Konrad Wieland. *Model Versioning and Enterprise Architect*. PowerPoint Presentation, LieberLieber Software GmbH. July 2017.

[YC17] Bobbi Young and Paul Clements. "Model Based Engineering and Product Line Engineering: Combining Two Powerful Approaches at Raytheon." In: *INCOSE International Symposium* 27.1 (July 2017), pp. 518–532. DOI: 10.1002/j.2334-5837.2017.00376.x.

[ZDB16] Bo Zhang, Slawomir Duszynski, and Martin Becker. *Variability Mechanisms and Lessons Learned in Practice*. Tech. rep. Fraunhofer Institute Experimental Software Engineering (IESE), Kaiserslautern, 2016. DOI: 10.1109/VACE.2016.012.

[Zur14] Gabriela Zurbuchen. "A Case Study for Integrated Lifecycle and Variant Management in a SME Context." MA thesis. TU Kaiserslautern, 2014.

[pur20] pure-systems GmbH. *pure::variants User's Guide*. v5.0.0.685. 2020.

DECLARATION

Hereby, I declare that I have composed the presented thesis independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

Freiburg i. Br., August 2020

Florian Rohlf