

Masterarbeit

Evaluation of variant management capabilities of automotive software engineering tools

Nermin Kajtazović

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß



Begutachter: Dipl.-Ing. Dr. techn. Christian Kreiner
Betreuer: Dipl.-Ing. Andrea Leitner

Graz, im März 2011

Kurzfassung

Durch die zunehmenden Wechselwirkungen zwischen der System- und Softwareentwicklung von eingebetteten Systemen in Hybridfahrzeugen werden die Systemfunktionen immer mehr softwaretechnisch realisiert. Außerdem versuchen die Fahrzeughersteller und die Zulieferer ihre Produkte möglichst flexibel zu gestalten, um aus ihren Entwicklungsumgebungen unterschiedliche Fahrzeugvarianten zusammenbauen zu können. Diese Varianten wirken sich auch direkt auf die Software aus.

Die heutzutage angewandten Softwareentwicklungsansätze stellen sich als nicht ausreichend heraus, weil durch die Neuimplementierung von Software für jede neue Variante hohe Entwicklungskosten, sowie stark beschränkte Wiederverwendbarkeit der bereits existierenden Funktionalität entstehen.

Um diesen Problemen zu entgehen, wird im Rahmen des Projektes HybConS (Hybrid Control System) eine generische Lösung, basierend auf strategischer Wiederverwendung von Software, angeboten. Die Idee dahinter ist es, eine Basis für die systematische Wiederverwendung und die Verwaltung von allgemeinen und produkt-spezifischen Softwareartefakten aller Phasen des V-Models zu schaffen.

Das Ziel dieser Masterarbeit ist es, so eine Basis für die Softwarearchitektur zu realisieren. Dabei wird die Umsetzung der Strategie von einem passenden Ansatz zum Variantenmanagement (Software-Produktlinien (SPL)) unterstützt. Die Entwicklung der generischen Architektur, sowie ihre Wiederverwendbarkeit, sind in SPL durch zwei getrennte Prozesse realisiert. Die beiden Prozesse wurden für den praktischen Teil dieser Arbeit als ein Prototyp entwickelt. Außerdem wurden einige Architekturbeschreibungssprachen (ADL) analysiert und anschließend wurde eine Evaluierung von Werkzeugen durchgeführt. Das Ziel dieser Evaluierung war, passende Werkzeuge zur Unterstützung von SPL-Prozessen zu finden. Die Evaluierung resultierte in einer integrierten Werkzeug-Umgebung, zusammengesetzt aus pure::variants und Papyrus. Besonders intensiv wurde auf die Spezifikation von EAST-ADL eingegangen, welche zur Beschreibung der generischen Architektur verwendet wurde. Diese Architektur dient zur Dokumentierung des integrierten Systems.

Das EAST-ADL Modell wird durch das Extrahieren struktureller Informationen aus der Softwareimplementierung aufgebaut. Zur Beschreibung von Softwarekomponenten, wird in diesem Projekt AUTOSAR eingesetzt. Der technische Hintergrund für diesen Generierungsprozess ist die Modelltransformation, die das Mapping zwischen AUTOSAR und EAST-ADL Modellen unter Berücksichtigung einer vordefinierter Mapping-Strategie realisiert. Zusätzlich wird die Variabilität basierend auf der Spezifikation von EAST-ADL generiert. Zusammenfassend zeigt dieser Ansatz eine mögliche Integration von EAST-ADL in einem Softwareentwicklungsprozess in der Fahrzeugtechnik. Die als Abschluß der Arbeit durchgeführte Evaluierung stellt die erzielten Ergebnisse bzgl. der Erwartungen an die Methodik zur Schau.

Abstract

The system development in the automotive domain nowadays is confronted with a high increase of software functions. Moreover, OEMs (Original Equipment Manufacturer) as well as suppliers are trying to offer highly flexible products by deriving different configurations from their development infrastructure. These variants have a direct influence on the software. Implementing each variant individually leads to an enormous increase of development costs and low reusability of core assets (parts of the system).

The main idea proposed in the HybConS project (Hybrid Control System) to address these issues is to provide a common base for strategic reuse of the control software and to manage product-specific core assets (variabilities) in all phases of the V-Model.

The aim of this thesis is to provide such a reuse strategy for architectural core assets. The approach of choice for variant management is a software product line (SPL). The development and the reuse of the generic architecture in SPL are driven by separated engineering processes. In the practical part of this thesis a prototypical implementation of these processes for the automotive context is provided. Moreover, the analysis of architecture description languages (ADL) for their applicability in the automotive domain and a tool evaluation are performed. The aim of this tool evaluation was to find an adequate tool or tool chain to support the SPL engineering processes. For this project, an integrated tool environment consisting of pure::variants and Papyrus has been chosen.

Main focus concerning the implementation of the prototype is given to the EAST-ADL specification, which is in this project used to describe the software architecture. The purpose of this architecture is to document the integrated system.

The EAST-ADL model is created by extracting the structural information from the implementation of the control software, i.e. from software components. In this project, software components are formally described with AUTOSAR. The technical background for this generation process is the model transformation. It uses the predefined mapping strategy to transform software components described with AUTOSAR into an EAST-ADL Functional Design Architecture (FDA). In addition, the variability logic for handling of the architectural development artefacts is generated. It follows the EAST-ADL specification for variant management.

This approach, in summary, shows a possible integration of EAST-ADL in the automotive software development process. Finally, the achievement of main expectations on this methodology is evaluated.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

.....
date

.....
(signature)

Acknowledgment

This master thesis has been carried out at the Institute for Technical Informatics, Graz University of Technology.

At this point, I would like to thank my supervisors Andrea Leitner and Christian Kreiner for their great support and willingness to take time for many discussions that helped me to stay on the right track all the time. Further, I would like to thank Mario Driussi and Florian Pözlbauer from the Virtual Vehicle Competence Center (ViF) for the provision of evaluation use cases.

At last, I would like to thank my family and Belgin.

Graz, in March 2011

Nermin Kajtazović

Contents

1	Introduction	21
1.1	Problem Description	21
1.1.1	Project Description	21
1.1.2	Project Goals	22
1.2	Thesis Structure	23
2	Related Work	25
2.1	Systematic Reuse of Software	25
2.1.1	Software Product Line Engineering	25
2.1.1.1	Principles and Motivation	25
2.1.1.2	Domain Engineering	27
2.1.1.3	Application Engineering	29
2.1.1.4	Documenting Variability	30
2.1.1.5	Terminology	32
2.1.1.6	Application of SPLE in Software Engineering	33
2.1.2	Variant Management in the Automotive Domain	36
2.1.2.1	Introduction	36
2.1.2.2	Application of Product Lines in the Automotive Domain	37
2.1.2.3	OEM-Supplier Relation in context of Product Lines	38
2.1.2.4	Classification-based Approach for Variability Modeling	38
2.2	Software Architecture for Automotive Systems	39
2.2.1	Specification of Software Architecture	40
2.2.2	Architecture Description Languages and Standards	41
2.2.2.1	AADL	42
2.2.2.2	SysML	42
2.2.2.3	AUTOSAR	43
2.2.2.4	Fibex	53
2.2.2.5	EAST-ADL	54
2.3	Model Transformation	59
2.3.1	Model Transformation Approaches	60
2.3.1.1	Model-to-Text Transformation	60
2.3.1.2	Model-to-Model Transformation	60
2.4	Tool and Language Evaluation	61
2.4.1	ADL Selection and Evaluation	61
2.4.2	Tool Selection and Evaluation	63
2.4.2.1	Selection Criteria and Prerequisites	63

2.4.2.2	Evaluation Methodology and Tools	64
2.4.2.3	Results of Tool Evaluation	76
2.5	Hypothesis	77
3	Design of the HybConS Architecture	81
3.1	Requirements	83
3.2	Scope	84
3.3	Domain Engineering	84
3.3.1	Variability Documentation	84
3.3.1.1	Variability in Design Assets	86
3.3.1.2	Binding Times	86
3.3.2	Product Management	86
3.3.3	Software Component Description	87
3.3.4	Mapping Strategy	90
3.3.4.1	Existing Approaches	90
3.3.4.2	Concept	92
3.3.5	Variability Model	93
3.3.5.1	Analysis of Variant Management in EAST-ADL	94
3.3.5.2	Variant Management in HybConS	97
3.4	Application Engineering	99
3.5	Final Architecture	100
4	Implementation of the Prototype	103
4.1	Domain Engineering	104
4.1.1	Part I: Mapping Process	104
4.1.1.1	Development View	104
4.1.1.2	Process View	106
4.1.2	Part II: Variability Extension Generator	110
4.1.2.1	Artifact Level Variability	110
4.1.2.2	Vehicle Level Variability	111
4.2	Application Engineering	113
4.2.1	Part III: System Configuration	113
4.2.1.1	VSL Expression Evaluator	114
4.3	Integration into the HybConS Tool Environment	115
4.4	Technology	116
5	Evaluation	117
5.1	Methodology	117
5.2	Use-Cases	118
5.2.1	Use-Case1: Simple Vehicle Hybrid System	118
5.2.2	Requirements on SPL	119
5.2.3	Solution	119
5.2.4	Use-Case2: Seat Adjustment in Vehicle System	120
5.2.5	Requirements on SPL	120
5.2.6	Solution	121
5.3	Results	121

5.3.1	Reusability	123
5.3.2	Scalability	123
5.3.3	Prototype Evaluation	124
5.3.4	Performance Analysis	124
6	Conclusion	127
6.1	General	127
6.2	External vs. Internal Variability	128
6.3	Plain Propagation	129
7	Future Work	131
7.1	Mapping Process	131
7.1.1	Model Transformation	131
7.1.2	Behavior Mapping	131
7.2	Variability Extension	132
7.2.1	External System Configuration	132
7.2.2	Formula Expression	132
7.2.3	Attributes, Associations and Property Sets	133
7.3	Diagram Information	133
A	Appendix	135
A.1	Tool Evaluation Criteria	135
A.2	Extension Guide	138
A.2.1	Interfaces	138
A.2.2	Feature Extension	140
A.2.2.1	Adding Model Elements	140
A.2.2.2	Adding Variants	141
A.2.2.3	Adding AUTOSAR XML Schema	142
A.2.2.4	Adding EAST-ADL Metamodel	142
A.3	Mapping Details	143
A.3.1	AUTOSAR	143
A.3.2	EAST-ADL	145
A.3.3	Mapping	147
A.3.4	Implementation Status	148
A.4	Evaluation Use Cases	150
	Bibliography	152

List of Figures

1.1	HybConS concept: a generic software development process	22
2.1	Motivations for product line engineering: development costs (left) and time-to-market (right), [PBvdL05]	26
2.2	Software product lines framework, [PBvdL05]	28
2.3	Information flows between application and domain engineering with respect to the stakeholder, [PBvdL05]	29
2.4	Variability pyramid, [PBvdL05]	30
2.5	Graphical notation for variability models, [PBvdL05]	31
2.6	Core process for system and software development in the automotive domain (V-Model), [SZ10]	36
2.7	Logical (left) and technical (right) abstraction levels of the vehicle architecture, [Kol06]	38
2.8	Development of the technical architecture: analysis of the control system, [SZ10]	40
2.9	Hardware/software interface: conventional (left) and AUTOSAR (right) [tre09]	43
2.10	AUTOSAR approach (left) and AUTOSAR ECU Software Architecture (right) [KF09]	44
2.11	Example: warn lights system realized in AUTOSAR, [KF09]	48
2.12	Development of ECU software based on AUTOSAR methodology, [KF09]	49
2.13	AUTOSAR XSD generator, [AUT09a]	51
2.14	FIBEX XML schema structure, [ZS07]	53
2.15	EAST-ADL domain structure, [CFJ ⁺ 10]	54
2.16	Product decision model, [ea08]	57
2.17	Bridge between two levels of variability in EAST-ADL	58
2.18	Concept of model transformation, [CH06]	59
2.19	EAST-ADL modeling workbench architecture, [SN10]	66
2.20	EAST-ADL package as a part of UML profile extension imported in MagicDraw	73
2.21	Interface connecting CVL and DSL models, [Cv1]	74
2.22	Fragment substitution in CVL, [Cv1]	75
3.1	HybConS processes reflected in SPLE framework	81
3.2	EAST-ADL functions interaction on design level	88
3.3	Graphical notation of AUTOSAR software component, [AUT09b]	92
3.4	Information flow in AUTOSAR [AUT09b]	93

3.5	Example of the product line in EAST-ADL: merging and optionality approach	95
3.6	Transfer function of a single rule in HybConS variant management mechanism	99
3.7	Design flow of domain sub-process for HybConS architecture	101
4.1	Simplified processes for architecture generator in HybConS	103
4.2	HybConS architecture generator: development view	105
4.3	Mapping process	106
4.4	Generation process for variability extension	111
4.5	Effect of multiple instances in feature models	112
4.6	System configuration process: configuration (left) and derivation (right)	113
4.7	Architecture plug-in: integration in a tool environment	115
5.1	Evaluation use case 1: read battery charge status in a simple hybrid vehicle system	118
5.2	Evaluation use case 2: seat adjustment, [AUT09c]	120
5.3	Evaluation results: development costs for SPL and single systems	122
5.4	Simple GUI for prototype evaluation	123
5.5	Runtime	125
6.1	Variability pyramid in HybConS architecture: ideal (left) and real (right) amount of external and internal variability	128
7.1	External system configuration with the FeatureMapper plug-in	132
A.1	Evaluation use cases	150
A.2	Evaluation use case 1: subsystems (soft real-time (blue) and hard real-time requirements from Section 5.2.2)	151

List of Tables

2.1	Different terminology for software product line, [Nor08]	33
2.2	Evaluation results for architecture description languages for the automotive domain	61
2.3	Papyrus advantages and drawbacks	67
2.4	MetaEdit+® advantages and drawbacks	69
2.5	pure::variants advantages and drawbacks	70
2.6	oAW advantages and drawbacks	72
2.7	Results of tool evaluation	78
2.8	Results of tool evaluation derived from Table 2.7	79
3.1	EAST-ADL2 elements used in FDA	89
3.2	AUTOSAR VFB elements related to Table 3.1	91
3.3	Deviation set in HybCons	100
4.1	Possible definitions of reference base	109
5.1	Runtime	124
6.1	Realized prototype features with respect to variant management	127
A.1	Criteria attributes for tool evaluation and selection	138
A.2	API documentation for the class <i>FeatureDescription</i>	139
A.3	Analysed AUTOSAR VFB metamodel elements	145
A.4	Analysed EAST-ADL metamodel elements	147
A.5	Detailed mapping between AUTOSAR VFB and EAST-ADL FDA (Source ID - AUTOSAR model element id, Target ID - EAST-ADL model element id)	148
A.6	Implementation status for AUTOSAR part of the mapping process	149
A.7	Implementation status for EAST-ADL part of the mapping process	149

Abbreviations

3GL	Third Generation Languages
AADL	Architecture Analysis & Design Language
ANTLR	Another Tool for Language Recognition
AOB	Aspect-Oriented Programming
API	Application Programming Interface
ATESST	Advancing Traffic Efficiency and Safety through Software Technology
ATL	Atlas Transformation Language
AUTOSAR	..	Automotive Open System Architecture
BSW	Basic Software
CAN	Controller Area Network
CCL	Context-Based Constraint Language
CVL	Common Variability Language
DSL	Domain-Specific Language
DSM	Domain Specific Modeling
E/E	Electronic/Embedded
EAST-ADL	..	Electronics Architecture and Software Technology Architecture Description Language
ECU	Electronic Control Unit
EMF	Eclipse Modeling Framework
FAA	Functional Analysis Architecture
FDA	Functional Design Architecture
Fibex	Field Bus Exchange
FIFO	First-In-First-Out
FM	Feature Model
FODA	Feature Oriented Domain Analysis
GOPRR	Graph Object Property Port Role Relationship
GP	Generative Programming
GUI	Graphical User Interface
HDA	Hardware Design Architecture
IA	Implementation Architecture
LIN	Local Interconnect Network
M2M	Model to Model
MAENAD	...	Model-based Analysis & Engineering of Novel Architectures for Dependable Electric Vehicles
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MDA	Middleware Design Architecture
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering

MERL	MetaEdit® Reporting Language
MOF	Meta-Object Facility
MOST	Media Oriented Systems Transport
NVRAM	Non-volatile Random Access Memory
OEM	Original Equipment Manufacturer
OMG	Object Management Group
OVM	Orthogonal Variability Model
PFM	Public Feature Model
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query View Transformation
RAM	Random Access Memory
RIF	Requirements Interchange Format
RTE	Run-Time Environment
SOAP	Simple Object Access Protocol
SPL	Software Product Lines
SPLE	Software Product Line Engineering
SysML	System Modeling Language
TTCAN	Time Triggered Controller Area Network
UML	Unified Modeling Language
VFB	Virtual Function Bus
VFM	Vehicle Feature Model
VSL	Variability Specification Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema Definition

1. Introduction

This thesis describes an evaluation of variant management in architecture description languages (ADL) with the focus on the automotive domain and proposes a proof-of-concept for systematic reuse and variant management of architectural core assets (structural elements) for a control software applied in hybrid vehicles. In addition, it describes an integration of the proof-of-concept into an existing tool environment.

1.1. Problem Description

The development of (hybrid) vehicle systems is aware of a high increase in complexity and a subsequently increase of provided functionality. This has as a consequence a dominant coverage of the software with respect to functions allocation (cf. [SZ10]: a number of software components in a vehicle system nowadays is of a three-digit). Besides this, the development of hybrid vehicle systems tends to provide high flexibility e.g. in order to easily fabricate various topologies of the power-train. Moreover, each topology is further configurable to meet individual needs through mass customization. However, similar products as a whole are not highly scalable for application of such a strategy (e.g. a single vehicle type has just a few different realizations), but the more important aspect is to share common functionality among various products. This claims to provide a way for systematic reuse of an existing software as well as the ability to customize it for specific needs. An appropriate approach to support this are software product lines (SPL). It defines two engineering processes, i.e. domain and application engineering. In the scope of the domain engineering process the platform containing reusable core assets is developed. The reuse of this platform is part of the application engineering process.

Another challenge for the strategic reuse of the hybrid software is its applicability in the currently used V-Model for system development. This process is shared between a supplier and an OEM (Original Equipment Manufacturer - a customer in this context) in such a way that development on the system level is conducted by the OEM whereby the development of ECUs (Electronic Control Unit, i.e. a part of the system) is in responsibility of a supplier. Now, applying a methodology for strategic reuse would have an impact on the whole development model and therefore on their relation too. The strength of dependency between them is in this case directly influenced by the application of the product line approach. This is a major difference to other domains and should be considered in the technical realization of the reuse strategy.

1.1.1. Project Description

The contribution of this thesis is an excerpt of the HybConS project (Hybrid Control System) whose development is in a cooperation between the Institute for Technical Infor-

1. Introduction

matics and two industrial partners, Virtual Vehicle Competence Center (ViF) and AVL List GmbH. The aim of the project is the provision of the *generic* software development process, which for a given product specification and an existing model repository generates the software product with all system artefacts, i.e. requirements, hardware/software architecture, implementation, and a test and a cosimulation environment in a given scope (see Figure 1.1).

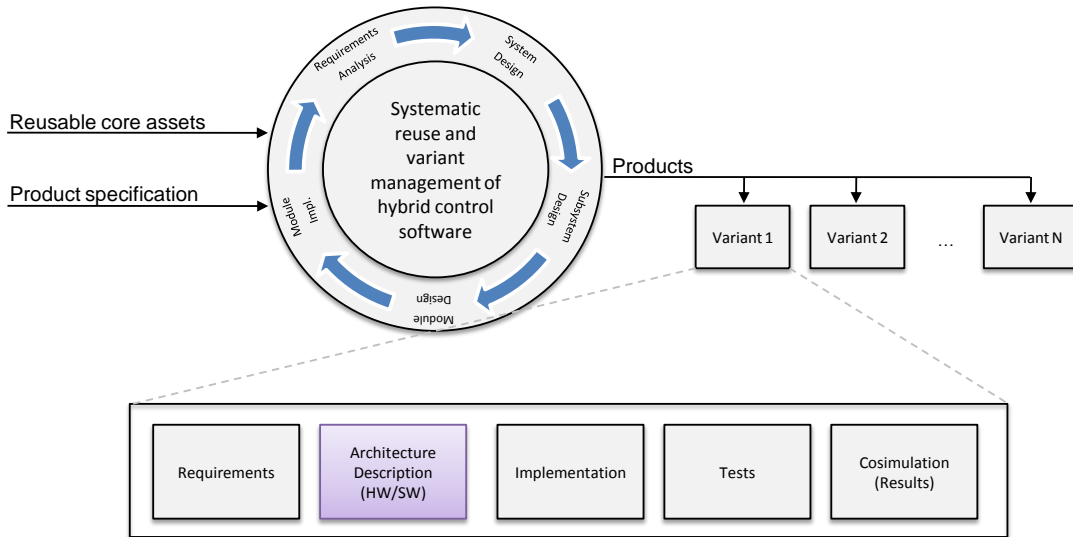


Figure 1.1.: HybConS concept: a generic software development process

This implies the handling of variants and systematic reuse in each development phase. Such an approach is provided by a software product line engineering framework (see Section 2.1.1). In the scope of the application engineering process the platform having the similar, but variant rich, structure is reused to generate a product (lower part of Figure 1.1).

Another challenge in this component-based development is the representation of the control software (reusable assets) in the platform. This requires an unique description of reusable assets at least for the domain implementation, which is afterwards used as a basis for the derivation of further core assets (architectural, test, and other assets).

As stated before, just an excerpt of the HybConS project is the mission of this thesis, namely, a part of the circle, shown in Figure 1.1, which provides the variant management and systematic reuse for architectural core assets. It results in the variability-free software architecture (highlighted block in Figure 1.1) which in addition may be used as a documentation for the implemented system.

1.1.2. Project Goals

The most important expectations from the HybConS project are high reusability of existing core assets and the ability to easily extend the existing functions or to add new ones,

i.e. a high scalability of the control software. In addition, goals of the SPL are implicitly covered: reduction of development costs, reduction of time-to-market and increase of software quality (see Section 2.1.1.1).

Regarding the strategic use of architectural core assets (product line architecture) in this project, the goals are the same as for the whole HybConS project. In addition, the consistency between related core assets should be guaranteed.

1.2. Thesis Structure

Chapter 2 covers the literature research addressing various approaches, methodologies and paradigm related to the topic of this thesis. It starts with the systematic reuse of software with the focus on software product lines and its application in the automotive domain. In **Section 2.2** a short introduction to the system development in automotive projects is given with the intent to show the essential development process phases towards software architecture. Furthermore, this chapter introduces several approaches for model transformation in **Section 2.3**. As a conclusion for related work, two evaluations are presented in **Section 2.4.1**. The intent of the first evaluation is to find an adequate tool supporting the well known engineering processes of software product lines (SPL), whereas the other one should provide the formal description of the software architecture. Finally, **Section 2.5** concludes the related work.

Chapter 3 discusses the main design decisions for the realization of the product line architecture (variability in the architecture). **Section 3.1** defines the scope of the proof-of-concept with respect to coarse-grained requirements given in **Section 3.1**. Further content of this chapter describes the detailed analysis of SPL engineering processes with a focus on the software architecture in the automotive domain. The aim of this analysis is a definition of the strategy for generation of domain core assets and their reuse. The result of the design phase is a logical view of the product line architecture given in **Section 3.5**.

In **Chapter 4** the implementation of the product line architecture with respect to the design described in the previous chapter is presented. Afterwards, the integration into the existing tool environment is briefly discussed.

Chapter 5 contains the evaluation results.

Chapter 6 concludes the work.

In **Chapter 7** the suggestions for further development of the proof-of-concept are given. They are a guide for a transition from the prototype to the product.

2. Related Work

2.1. Systematic Reuse of Software

Demand on systematic reuse of a large series of products is growing rapidly. In the last twenty years various methods and approaches addressing this domain have been developed and they are nowadays state of the art in many sectors of technology. The main reasons for such expansion are the reduction of production costs and better product quality. Actually, the origin of this idea was a result of a competition of companies on the market. In the late eighties Kodak has developed the first infrastructure for the production of cameras in a more systematic way in order to win market share against Fuji. The development process was based on a so called *common platform* combined from parts that are used in all different variants of cameras. This resulted in faster development and lower costs. Thus, in the nineties Kodak won the battle against Fuji and controlled about 70% of the US market, [PBvdL05]. This was inspiration for other companies (not only for these kinds of products) and the era of product lines begun.

Applying this strategy in the software led to a new transition in reuse history, [Nor08]. The software products are not handled in an opportunistic way anymore, but instead, they are combined from already existing parts which are extracted from a so called *software platform*. The platform in this context is a kind of repository from which a set of different products can be derived. The combination of the platform and the *mass customization*, i.e. production of different variants of products for individual needs, introduces a new software development paradigm for systematic reuse of software, which is in [PBvdL05] known as *Software Product Line Engineering*. However, some companies are still using opportunistic approaches like configuration management (CM) to handle variants in their products. For some kinds of projects this is a useful solution, but for others it is not. [DB07] discusses issues when managing product variants by using *Clone-and-Own*, *Independent Component Teams* and *Platform Version* approaches.

2.1.1. Software Product Line Engineering

Software product line engineering is in [PBvdL05] defined as follows:

... a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization.

2.1.1.1. Principles and Motivation

Two main principles of SPLE are mass customization and the platform. Mass customization can be seen as a complement to the platform. Without mass customization the platform would not be sufficient to handle software variants systematically, because it

2. Related Work

would not address individual customer needs. The first step in the creation of the platform is actually the preparation for mass customization. Here, not only the scope of the platform is defined, but also all characteristics of variants satisfying individual customer needs within this scope. Then, in the next step the common and variable parts of the software are defined. This has an impact on flexibility of the product line (range of products sharing the same platform). For instance, to handle four different variants of the product, only the parts specific to individual products may be defined as variable, i.e. they correspond to differences between products within the scope. Furthermore, these differences may be related to other common or variable parts of the product line (e.g. by hierarchical or sequential relations). This constraints the flexibility in the platform, which means that the platform for n variable parts does not produce 2^n products.

Applying software product lines is not only a challenge for the software developers, but also for the whole company. A company needs to adapt its organizational structure in order to provide the platform. To handle variabilities and commonalities in a systematic way, a company needs to provide strategies and technology to realize this. The products are not independent anymore. Instead, they are a part of the main generator, i.e. the platform.

Motivations for Software Product Line Engineering The main expectations on SPLE are the reduction of development costs, reduction of time-to-market and the enhancement of quality. Figure 2.1 shows these expectations graphically.

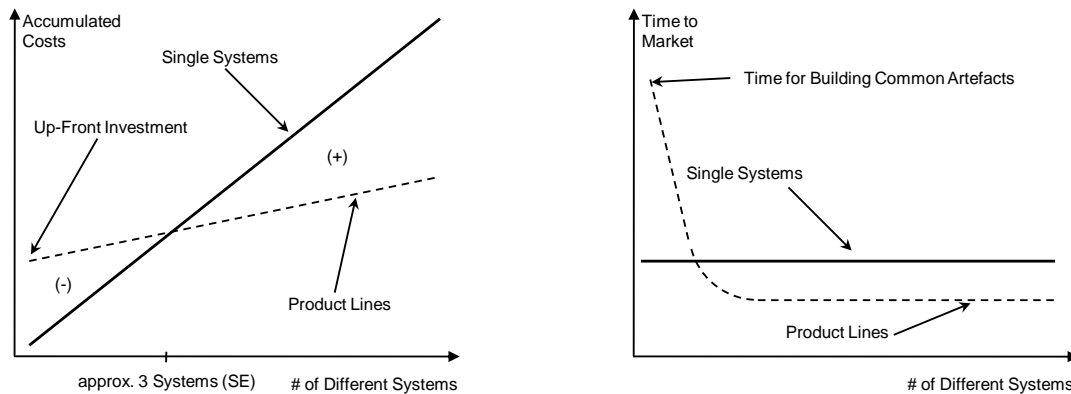


Figure 2.1.: Motivations for product line engineering: development costs (left) and time-to-market (right), [PBvdL05]

The left diagram shows the development costs for an increasing number of different products. Here, the conventional software development approach (a single-product approach) is compared to SPLE. Its line is idealized, i.e. it is assumed that each new product requires the same effort to be realized. From this idealization it follows that for the realization of n products $n * t_s$ time is required (t_s - time invested for development of a single product, if the time is assumed as a cost parameter).

Unlike a single-product approach, the SPLE starts with some up-front investment which is necessary to develop the platform. Further product realization is faster, but the generation of a couple of products is required in order to reach the *break-even point*, i.e. pay-off point. Therefore, it should first be decided if SPLE is an adequate approach for this certain project or not. The up-front investment depends on the transition to the product line. [HKM06] proposed a transition strategy, which, for a large development company, achieves the break-even point two orders of magnitude lower than typical incremental transitions which requires about 2-3 products to reach the pay-off. For such cases where only 2-3 single products require an effort of dozens of development months, it is difficult to apply the SPLE approach. Therefore, transition methods are the next research topic.

As shown in Figure 2.1 there is a slight increase of the SPLE line. This is due to an additional effort required to satisfy all requirements. For a given set of requirements the platform is used to build the product conforming to these requirements. But, it is not always the case that all requirements are satisfied. In this case, the product is derived according to the requirements, as far as possible. The remaining requirements have to be realized by applying a conventional software development process. It further has to be decided if the additional parts should become part of the platform.

The other diagram in Figure 2.1 compares the time-to-market for the SPLE and the single-product development approach. At the beginning of the SPLE approach, much more time is required, because the product line architecture needs to be built.

The higher quality in the SPLE is achieved through reuse. The parts need to be implemented and tested only once, and every further reuse ensures their proper functioning. Besides mentioned and most important motivation factors in SPLE, there are also other expectations like benefits for customers, for example the configuration of products on their needs, the improvement of cost estimation by using the platform, and the reduction of complexity, etc.

Now it is time to clarify how the platform is realized technically and what is behind the mass customization in this context. [PBvdL05] proposes the SPLE framework¹ which separates the platform and mass customization. These two principles are described by domain and application engineering processes (see Figure 2.2). Both processes are containers for software artefacts that are strongly dependent on the underlying domain. Software artefacts are distributed over various development phases forming the product line on the upper part of Figure 2.2 and a product below. The transition from domain to application engineering is the process of product derivation which collects necessary artefacts and builds the product satisfying given requirements.

2.1.1.2. Domain Engineering

Domain engineering is the process for building the platform. It starts with the product management which refers to economic aspects like defining the scope of the product line.

¹derived from ITEA projects: ESAPS, CAFÉ and FAMILIES, [PBvdL05]

2. Related Work

Within this scope, common and variable features for the next product range are specified. In [KCH⁺90] the feature is defined as follows:

A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems.

Since the whole process is iterative, later adaptations in this sub-process are allowed. The result of the product management is a product roadmap, i.e. the major specification of the product line. All other domain sub-processes are realized in correspondence to this specification.

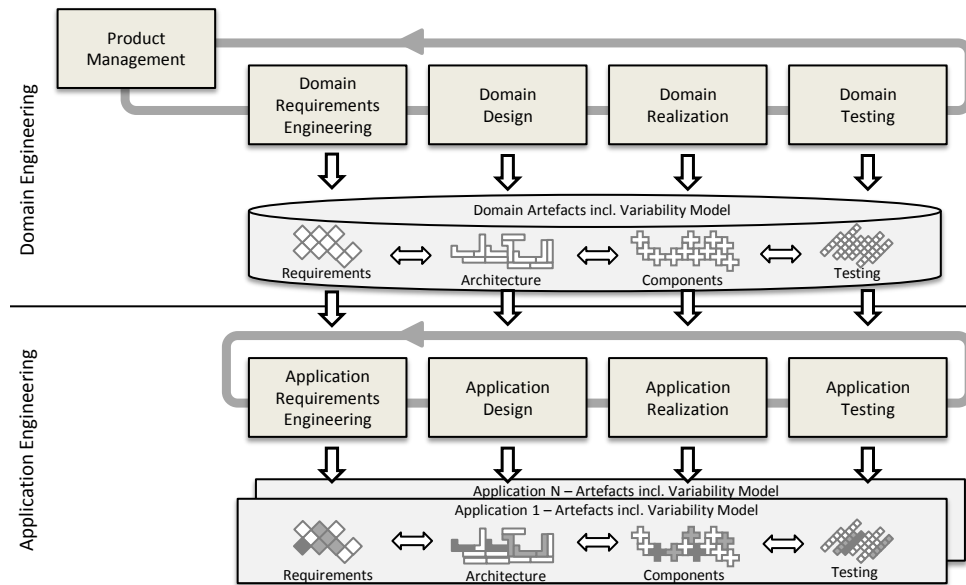


Figure 2.2.: Software product lines framework, [PBvdL05]

The aim of the requirements sub-process is to handle common and variable requirements expressed in various forms (e.g. textual, model-based, etc.). In correspondence to the product roadmap this sub-process produces the variability model of the product line which is then a main subject for product derivation (configuration).

In the next stage (domain design) the variability model is refined. Features (abstract characteristics of a system, or abstract requirements) within this model are refined to express variability in more detailed domain artefacts. In this case these artefacts are structural parts of the product line that form a so called „reference architecture“. Variable parts of the reference architecture are usually not visible to the customer or system developer who uses the platform. Instead, they are defined as „internal variants“ in order to hide details and to keep the variability model abstract enough. Moreover, these internal variants could be further decomposed to express more detailed configuration. In [RKW09] this kind of links between variants is known as *configuration hiding*. In this work, a method handling hierarchical variant dependencies is presented. It is explicitly handled in Section 3.

The transition to the next stage in the process follows the same schema as before: the reference architecture containing reusable common and variable architectural artefacts is an input to the realization phase. This sub-process results in a detailed system design and implementation. Similar to design, internal variability is further decomposed into a more detailed representation. From now, all domain artefacts are ready for validation and verification. This is performed by the testing sub-process. It is not able to test the whole application, because test artefacts are also covering the whole product line. Instead, a single implementation component as well as chunks of common artefacts can be tested. The purpose of this sub-process is to check whether the implementation conforms to requirements, architecture and design. Additionally, there is no overhead when implementing the tests for a single application.

2.1.1.3. Application Engineering

The main goal of application engineering is to achieve an as high as possible reuse of domain artefacts and the binding of variability in dependence of application needs (requirements to a single application). This is the location where the flexibility of the platform is reflected. A compromise between available platform artefacts and application needs has to be found. Furthermore, application needs may be extended by a stakeholder (e.g. customer), which complicates the situation, because the list of needs may not be satisfied by the platform (there is no product configuration which conforms to the specification by 100%). The difference between optimal configuration in the platform and the application needs is in [PBvdL05] termed as *delta* between domain and application requirements. The process of product pre-configuration is depicted in Figure 2.3.

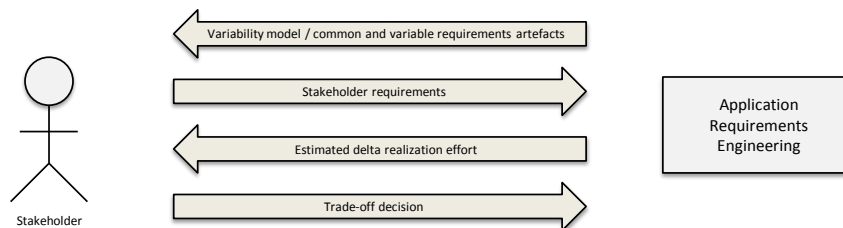


Figure 2.3.: Information flows between application and domain engineering with respect to the stakeholder, [PBvdL05]

Before configuring the product an additional implementation effort for fulfilling all requirements need to be estimated. This should ensure that costs caused by using SPLE are not too high, i.e. they must not reach the single-product costs. Therefore, a stakeholder should participate to pre-configuration activities. First, a stakeholder gets the variability model with all requirements artefacts in order to be aware of capabilities of the platform. These information help the stakeholder to adapt its needs which are in the next step delivered to the system (requirements) developer. In correspondence to stakeholder needs the effort for configuring the application which fulfils all requirements is calculated. Finally, a stakeholder decides if deltas should be realized or not. This is a reason for a slight increase of the SPLE line from Figure 2.1 (left).

2. Related Work

2.1.1.4. Documenting Variability

Concept of Variability Variability corresponds to any varying characteristic from a specific domain. It is a base for reuse of variable development artefacts. In a common language the variability is described by a variability subject and a variability object. To clarify this, for documenting variability three main questions need to be answered: (1) what varies, (2) why it varies and (3) how it varies. The first question refers to the subject (e.g. a language) with different levels of granularity (fine grained variant handling allows to provide detailed configuration, [KAK08]), whereas the answer to the third question corresponds to an instance of the subject (e.g. german). The second question refers to a reason why the subject *language* may have different realizations.

Classification In SPLE the variability subject is known as *variation point* extended by domain specific information and the variability object is defined as a *variant* (a single instance of the variation point). Variation points and variants may be classified by:

- Variant existence
Here it can be distinguished between variability in time and variability in space. The first kind corresponds to the existence of different realizations of the same “thing” (artefact) at different times. For instance, the variation point *Identification mechanism* developed in 1960’s would not be able to aggregate the variant *RFID identification*, but nowadays it would be possible. Unlike this, variability in space corresponds to the existence of different realizations of the artefact at the same time.
- Customer visibility
To allow customers to realize their needs, a part of the platform need to be visible for this group of stakeholders. As previously mentioned, variants (features) defined in domain requirements are further decomposed down to the implementation and testing. For a customer, just an abstract representation of the configurable parameters (features) is meaningful. Therefore, an excerpt of the variability model is combined from external features (visible to the customer) and the remaining parts are accessible to system developers only.

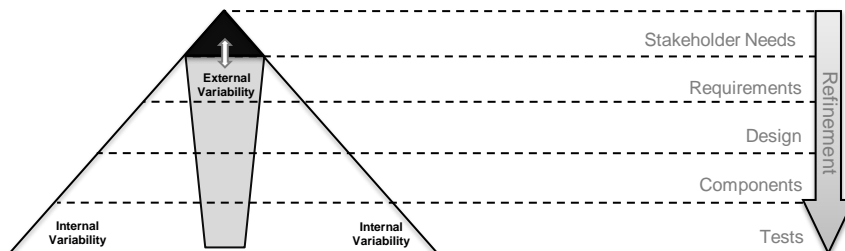


Figure 2.4.: Variability pyramid, [PBvdL05]

Variability Pyramid Distribution of variability in different abstraction levels is in SPLE reflected by a so called variability pyramid. It describes a typical amount of variability on each abstraction level. This is illustrated in Figure 2.4. The amount of variability increases

unproportional to the abstraction (e.g. requirement affects various design artefacts). However, more interesting is the distribution of external variability. External variability exists in implementation artefacts too, but in lower amount as in requirements. The reason is that a customer is more confronted with an abstract representation, but anyway some details from lower level artefacts may be visible.

Orthogonal Variability Model Variability of the product line is defined in a separated model called *orthogonal variability model* (OVM). This separation reduces the complexity of handling variants and keeps the artefact model as simple as possible. The graphical notation of the OVM elements is depicted in Figure 2.5. As mentioned before, the number of possible products is constrained by dependencies between variants. As shown in Figure 2.5, it is possible to “require” another variant if the current one is selected or to “exclude” a certain variant. In addition, the hierarchical dependencies *mandatory* and *optional* may be also defined. Optional dependencies may be further specialized to alternative (XOR),

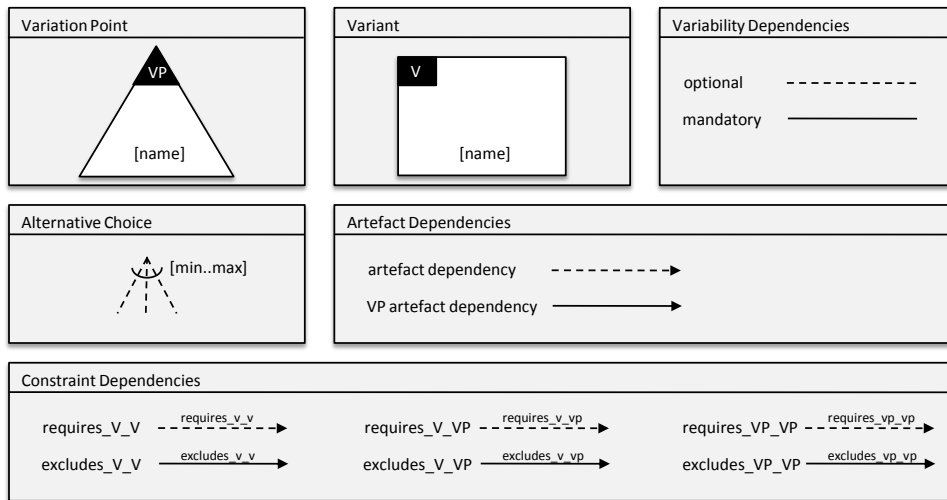


Figure 2.5.: Graphical notation for variability models, [PBvdL05]

i.e. only one variant may be bound.

Variability in Requirements Requirements may be described in various ways. For the introduction of variability in requirements it is assumed that they are in form of models or textual representation. The first describes requirements as optional and alternative features in a so called *feature model* or *feature tree*. This is similar to the OVM and typically is used as an abstraction to the OVM. The other form of artefacts would be a flexible textual representation of requirements, i.e. parts of the text are variation points providing several textual alternatives. The textual representation is more difficult to handle than requirements represented as models.

Variability in Design Variability in design corresponds to variation points and variants of the reference architecture. To handle this, three views of the architecture are considered:

2. Related Work

(1) development view, subdivided into subsystem and layers, components, interfaces and configurations, (2) the process view and (3) the code view. Variability in these views is discussed in Section 3.3.1.1. An example of the reference architecture from the component view is depicted in Figure 3.5.

Orthogonal to these views, [BB01] describes the following sources for variability in the architecture:

- Function - This variation point deals with the existence of a single function.
- Data - Variable data structure (e.g. attribute variation points in AUTOSAR, see Section 2.2.2.3).
- Control flow - Various representations of the control flow. This can be handled in the process view of the architecture.
- Technology - This includes variations of sensors, hardware, platform (not the product line platform), etc.
- Quality goals - Variation in quality (e.g. performance) is directly affected by variations in previous sources (e.g. applied technology behind the variable control flow may affect the performance).
- Environment - Variation in the interface exposed to the environment may also vary.

Variability in Realization The reference architecture is a basis for the variability in realization (implementation). A rough architecture variability is further decomposed in order to provide variability in detailed design first. Domain realization allows the following locations to be potential variation points: (1) component interfaces, (2) algorithms, (3) resources, (4) application configuration and (5) components and its parts.

Variability in Testing Systematic reuse of test artefacts means to handle test plans, test cases, test case scenarios, scenario steps and the report in the OVM. A test plan is a container for test cases. Each test case is further a container for scenarios describing different ways for goal achievement. The goal, an input, an expected output and a condition are parts of the same test case.

2.1.1.5. Terminology

In the previous sections terms like „software product lines“, „domain engineering“, „application engineering“ etc. have been used. However, there is another series of synonyms for these terms, because conferences of the underlying software paradigm have been independently scheduled in Europe and America for a long time. Table 2.1 shows the different terminologies. Both conferences are since 2004 merged into one and form the leading software product line conference, [PBvdL05]. In coming sections a mixture of terms is used.

Terminology	
SPL	PLE
Product Line	Product Family
Software Core Assets	Platform
Business Unit	Product Line
Product	Customization
Core Asset Development	Domain Engineering
Product Development	Application Engineering

Table 2.1.: Different terminology for software product line, [Nor08]

2.1.1.6. Application of SPLE in Software Engineering

[EV05] gives a short overview about the most important software development methodologies and paradigm supporting software product lines. Afterwards, several methods for product line architectures collected in [Mat04] are introduced.

The methods are principal addressed to either (1) architecture-centric or (2) component-based approaches. A component-based approach of PL builds the platform (a reusable component framework) in a bottom-up manner, i.e. development by composition. Complementary to this an architecture-centric, top-down, approach. [OB02] tries to find a balance between the mentioned two approaches in order to easily handle the PL in a *product population* (large product range).

Domain Specific Language A DSL is a textual or a graphical language (in literature known as *micro-language*, [vDKV00]) developed to address a specific problem domain. Unlike general purpose languages like C and Java, it is more powerful because a smaller range of problems is handled. Examples of such languages are SQL, HTML, regular expressions, etc. (other examples can be found in [vDKV00]).

The development of the DSL starts with the analysis of the problem domain (scope). For all applications within the scope, a language grammar for their constructs and relations is specified. In the implementation phase a DSL is packaged as a library and the language generator (compiler or interpreter) is implemented. From now, DSL programs addressing domain can be implemented and compiled (or interpreted).

Realizing SPLE with a DSL implies the definition of the problem domain (analysis phase) with respect to reuse of its constructs, i.e. the grammar of the language should provide the configurable set of applications in contrast to the current, fixed set.

Generative Programming The principle of GP is the mapping between a problem space and a solution space. The problem space contains the specification, which is represented

2. Related Work

in form of features. For a given specification the system can be automatically generated (solution space). Furthermore, the problem space can be expressed by e.g. a textual DSL to describe the specification. In this case the DSL acts as a way to select features in the configuration process (feature selection, dependencies, construction rules, etc.). GP uses the transformation to map the system configuration from the domain into the system implementation from solution space, [Cza04].

Model Driven Engineering MDE is a discipline in software development with the purpose to generate the system from models. These models (their structure and relations, restrictions, etc.) conform to metamodels which are basically the same as a grammar in DSL (but in form of model elements). An example of such model-metamodel relation is UML 2.0 (Unified Modeling Language) and MOF 2.0 (Meta-Object Facility). In this case, MOF is a language (metamodel) which is used to develop UML (model). Thus, UML conforms to the MOF.

The difference to DSL is that the target model in MDE is more generic, i.e. it can be constructed by several models in different levels of abstraction, different views, etc. The concept of a domain knowledge is specified by formal meta-models and its specific realization is generated by using model transformations, [EV05]. This process is further handled in Section 2.3.

To realize SPLE with the MDE approach the problem and solution spaces have to be formalized, i.e. the representation of common and variable domain artefacts as well as features is essential. [EV05] demonstrates how a family-based approach (MÉLUSINE) can be realized with the MDE.

Domain Specific Modeling DSM is similar to generative programming, but here the analysis phase results in the model which describes the concept of the domain and not features like in GP. This model is then used to define the DSL. The remaining steps in the generation process are the same as in GP, [EV05].

The most expressive power of DSM is the specification of the solution space in an abstraction level beyond programming. This can be compared as a transition from assembler to third generation languages (3GL) which enabled the developers to easily understand the formalism and to solve more complex problems than before. In DSM on the first line, a solution space is visually modeled by using domain concepts. This corresponds to a high-level specification from which the system is generated. A generation is performed by domain-specific code generators. For this purpose DSM uses a domain-specific modeling language, a domain-specific code generator and a domain framework, [KT08].

Further information about code generation patterns can be found in [Pre10].

The statistics show that the productivity by using MDE in contrast to the general purpose development is in range of 300%-1000% higher (Nokia has reported 1000% of productivity increase, [KT08]).

COPA Component-oriented Platform Architecting Method for Families of Software Intensive Electronic Products - is a PLA (Product Line Architecture) method developed by Philips in the scope of the Gaudi project. It aims to find the balance between architecture-centric and component-based approaches in order to find an optimal calibration between process, architecture, business and organization, [Mat04].

For product derivation (known as *architecting* in [Mat04]), the COPA requires an input consisting of stakeholder needs (expectations), architecture and a intuition of an architecture (one of the three sources of the architecture, [FBC06]). The output is a lightweight architecture (def.: architecture with minimized weight performed by minimizing weight of each architectural rule - scope, implementation details, level of details etc., [Mul10]) conforming to the specification.

The goal of COPA is the flexible management of complexity and platform size in a business environment.

FAST The Family-oriented Abstraction, Specification and Translation process was introduced by David Weiss in the 1990's to handle multiple system versions by sharing their common functionality in an organization. It relies on the SPLE principle of separating the domain into domain engineering (investment), product engineering (payback) and domain qualification (similar to the product management in SPLE). The domain engineering process defines common and variable features for family models (cf. artefacts in SPLE) as well as a decision model which maps these two spaces. Product derivation is usually performed by a compiler or composer for AML (Application Modeling Language), [WL99].

FORM The Feature-Oriented Reuse Method is an extension of FODA (Feature-Oriented Domain Analysis) proposed by Pohang University of Science and Technology to support the reuse of architectural and implementation domain artefacts. Parts realized by the domain engineering process are the reference architecture (see Section 2.1.1.2), a feature model and reusable components on implementation level, [Mat04].

This approach is similar to problem and solution spaces in pure::variants (see Section 2.4.2.2).

KobrA KobrA - Komponenten**bas**ierte Anwendungsentwicklung (german expression for *component-based application development*) is a component-based PL approach developed by Fraunhofer IESE for modeling (reference) architectures by using a model-driven software development paradigm. The software engineering process in KobrA is separated into the framework (cf. domain engineering in SPLE) and application engineering, [Mat04].

QADA Quality-driven Architecture Design and Analysis is a quality-driven architecture design method developed at the Technical Research Centre of Finland. In contrast to previously introduced PL methods, QADA also includes quality requirements to build the reference architecture. Actually, quality requirements are a deciding factor to build the software structure. The QADA process results on one hand with conceptual and concrete

2. Related Work

architecture and on other hand with analysis results which act as a feedback indicating whether the quality requirements are satisfied, [Mat04].

2.1.2. Variant Management in the Automotive Domain

2.1.2.1. Introduction

Before starting with a domain specific variant management it is necessary to give a short introduction into the system development in automotive. The whole development process involves the vehicle, the software and the system development interacting with each other. This results from the fact that the origin of about 90% of innovations in the automotive domain is realized in software and vehicle electronics, [Kol06]. Therefore, an adequate system development model covering all sub-processes and development phases from system requirements to acceptance tests is required. This is achieved by the well-known V-Model which on the one hand separates the system development phases from the software and on the other hand provides quality testing activities. Thus, these are reasons for its wide acceptance in the automotive domain, [SZ10]. This model is depicted in Figure 2.6.

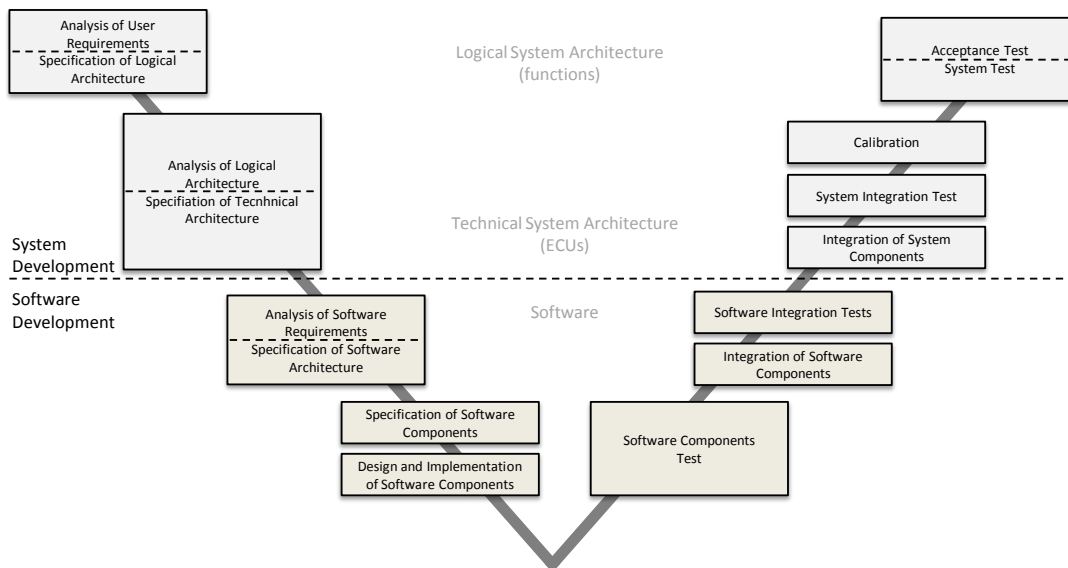


Figure 2.6.: Core process for system and software development in the automotive domain (V-Model), [SZ10]

The purpose of the first phase in the V-Model is to build the functional description of the whole vehicle or of the sub-system in correspondence to user requirements. This functional description (logical architecture) does not include any technical details, but just communicating functions and their interfaces.

In the next development phase the technical system architecture is created. It corresponds to the network of ECUs (Electronic Control Unit). Additionally, the functions from the previous phase are already partitioned and hence it is known which of them need

to be realized in the software. These functions are software requirements for the next phase.

In the lower part of Figure 2.6 (Software Development) the implementation of the software is provided. It starts with the analysis of the system functions allocated to the software and in correspondence to this function sub-network, the software architecture, i.e. system boundaries and interfaces, is created. In addition, software components may be conceptually specified. This concept is then in the design phase refined to details which are then used for the implementation of components.

After the implementation phase, the software components are integrated into a single software system. This integration is tested before the software is released. From now, the software is ready to be bound on the hardware. This is done by integrating system components into the whole vehicle system. Again, this integration is tested. Before the validation against user requirements is performed, the system functions are calibrated, e.g. for specific vehicle variants.

2.1.2.2. Application of Product Lines in the Automotive Domain

[Kol06] discusses different scenarios for the application of software product lines in the automotive domain. These scenarios are reflected by the following parameters:

- Application goals
- Application scope
- Development methods

Application goals describe what should be realized with the product line. Three possible options are proposed: (1) development, (2) configuration and (3) reconfiguration of vehicles/ECUs. The first is related to reuse of domain artefacts in order to build a new system. In the configuration, the product line is used just as a verification mechanism which checks whether a given vehicle configuration is feasible with the platform, i.e. if such a product (vehicle) can be derived. The last application goal is related to the replacement of existing components. This implies handling variability in time (see Section 2.1.1.4).

The application scope (parameter) deals with scenarios related to a specific domain where the product line may be applied (e.g. on ECU level only). These scenarios are related to both vehicle architectures (see Figure 2.7).

The logical architecture consists of functions logically grouped into function groups. Similar to this, the technical architecture is decomposed into three abstraction layers: the technical architecture as composition of sub-systems, the sub-system itself and the ECU. All these locations may be realized to represent the platform. However, it is very important to decide where to apply it, i.e. a decision should be beneficial for the OEM as well as for the supplier. The OEM's responsibility in the system is typically the upper part of the V-Model shown in Figure 2.6 (system development), whereas the supplier provides the software and the hardware of the ECU (lower part of Figure 2.6). Handling variability in technical artefacts (right side of Figure 2.7) introduces an additional overhead because of

2. Related Work

the presence of technical details on the one side and because of the hardware on the other side. Unlike this, functions are more abstract and better applicable for product lines. An issue is how to handle the OEM's and supplier's part from the abstract representation of the system. Practical experiences show that an ECU product line is the most applicable method. This comes from the fact that on the one hand the logic is fully decoupled from the OEM and on the other hand the product line ECU is not shared among suppliers, i.e. one ECU - one supplier.

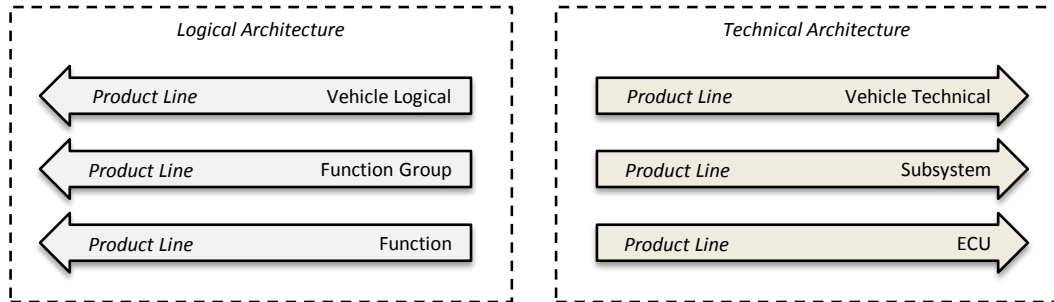


Figure 2.7.: Logical (left) and technical (right) abstraction levels of the vehicle architecture, [Kol06]

The last parameter (development methods) deals with the applicability of the product line methods with respect to the OEM-supplier relationship.

2.1.2.3. OEM-Supplier Relation in context of Product Lines

In [BK04] three different applications of product lines are analysed with respect to the OEM-supplier relation in order to find the optimal solution for both parties: (1) product line architecture handled by the OEM, (2) reuse of requirements and test-cases through abstraction and (3) binding the OEM's and supplier's product lines by an unified interface.

The first option is not practical, especially for the supplier, because the OEM delegates the process of the configuration. In addition, the supplier is more dependent on the whole process. The next option is much better. In this way, the supplier may adapt its product line with respect to given abstract requirements. This allows more independence between OEM and supplier. The third option is probably the best solution. As mentioned before, the OEM delegates the product line of the system description and the supplier the software product line of the ECU. The suggestion is that both parties make an agreement on specifying a common interface between these two product lines. This would be a single coupling between both parties. Moreover, it would not be just an abstract specification, but detailed.

2.1.2.4. Classification-based Approach for Variability Modeling

Nowadays OEMs are trying to reduce the complexity in development of E/E systems by adapting their organization processes to the AUTOSAR standard, [MA09]. This has

2.2. Software Architecture for Automotive Systems

enabled to switch to the function-based variability modeling in contrast to previously applied hardware-based feature modeling. The last describes the system in a feature model with traces to the *function network*. This function network acts as a platform. It contains all possible interconnections between hardware dependent functions and their traces to the real hardware. In application engineering deselected model elements are just removed from the function network and mapping to the hardware is done afterwards.

Another modeling approach allows to handle variants without the need to take care about the underlying hardware. This implies that the interconnection overhead in the function network is drastically reduced.

[Kae09] proposes a variability modeling approach by describing variants in three different abstraction levels: (1) feature level, (2) function level and (3) architecture level. The function level describes the system as a composition of functions. Here it is not yet decided which functions are realized in the software and which ones are realized in the hardware. Since AUTOSAR provides hardware-independent software modeling, all technical details from the function model are excluded. After partitioning, the platform is combined by architectures describing the software and hardware separately. The variation points on the third level are traced by configuration links to the feature models. Configuring automotive systems in mentioned three phases reduces complexity of the configuration and of variability modeling. Moreover, the absence of technical details in the functional description allows to easily implement changes in the product line.

The variant management introduced in [SZ10] as *Configuration Management* relies on handling versions of components by providing variability and scalability mechanism in a network-based component hierarchy (tree-based hierarchy where components may be assigned to several systems). However, configurations are orthogonal to this hierarchy and such *flat* variant management does not seem to be really optimal solution (at least for automotive), because of the global system configuration (cf. AUTOSAR, see Section 2.2.2.3), [RKW09]. An alternative to this mechanism is the *compositional variability* described in Section 2.2.2.5.

2.2. Software Architecture for Automotive Systems

[BCK03] defines the software architecture as:

... the structure of structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

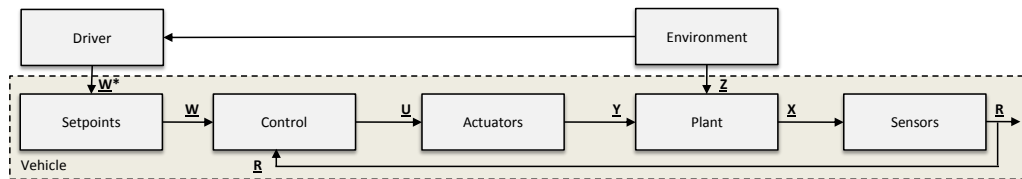
As already elaborated in Section 2.1.2.1, the software architecture in the automotive domain describes the system on the one hand as a composition of functions forming the logical system architecture and on the other hand as a network of interconnected ECUs. The aim of this section is to introduce the technical architecture with the focus on software functions.

2. Related Work

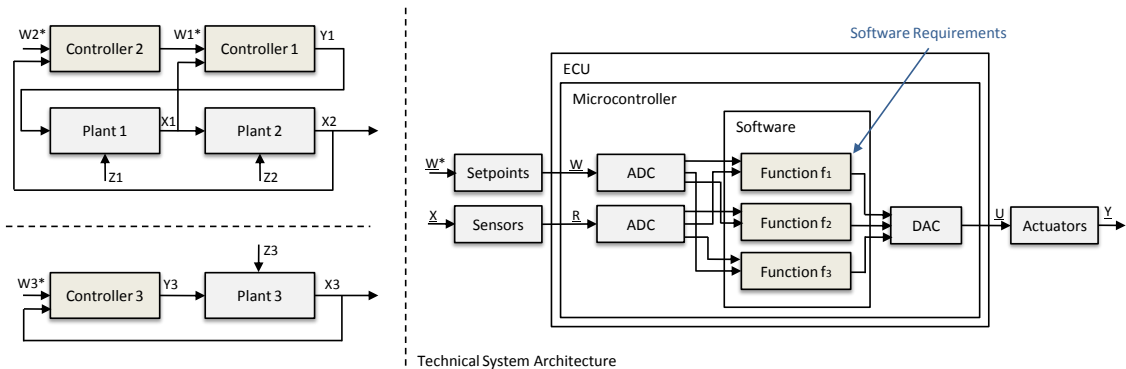
2.2.1. Specification of Software Architecture

The logical system architecture is built by realizing technical and non-technical system requirements. It acts as a binding level between requirements and the technical system architecture. Such an abstract system model defines what the system has and excludes all technical details on how the functions are realized. In further development steps the functions are allocated to concrete target subsystems and components inside the technical architecture. This task requires advanced analysis and specification methods to provide adequate realization with respect to existing system functions, because their realization highly depends on decisions and details provided in the technical system architecture. These methods may include analysis of control systems, real-time systems, distributed systems, reliable systems and other systems. The results of analysis provides various alternatives to realize functions from the logical system architecture, [SZ10].

From now, a solution for the technical system architecture is found. The functions from the logical architecture are allocated to either hardware components or software functions known as *software requirements*. To realize the software architecture, software requirements need to be extracted into a new view, which in addition defines the boundary of the software. Before going into detail in the development of the software architecture it is advisable to show how the software requirements are realized inside the technical system architecture. As previously mentioned, there are several analysis methods for function realizations. In the following, the technical architecture of a control system is described.



(a) Model of the control system in the logical architecture



(b) Exemplary control system in the logical architecture (left) and its realization in the technical architecture (right)

Figure 2.8.: Development of the technical architecture: analysis of the control system, [SZ10]

Figure 2.8(a) shows the model of the control system used in the analysis phase for the development of the technical system architecture. Many systems in a vehicle show the controlling character and hence the realization of their control functions is typically a part of the software. Therefore, these systems have a big influence on the system software. The model above shows the interaction between the vehicle system, a driver and an environment. The principle of such systems relies on controlling the plant by sensing the value X , comparing it to a given value W and adjusting their difference (error) with respect to a transfer function of the controlling block. The controlling process is finished when the difference is equals to 0. Assuming that a plant model is an Otto engine, the value W would be a fuel amount set by a driver (W^*). The controlling process tries to reduce the difference between X and W to 0 even the presence of the environment influence on the plant model (Z) disturbs the process (e.g. caused by diagnostics tests), [SZ10].

In Figure 2.8(b) the concrete usage of the mentioned model is presented. On the left side two control systems controlling the plant (e.g. engine) are modeled as a part of the logical system architecture. On the right side of the figure, their representation in the technical system architecture is proposed. As shown in the figure, the control functions (f_1 , f_2 and f_3) are realized in software². This software block represents the software requirements used in the next phase to build the software architecture.

The development of the software architecture relies on the analysis of software requirements and a specification of:

- Software components and their interfaces
Specification of on-board and off-board interfaces.
- Software layers
Specification of software layers in correspondence to their abstraction level. In AUTOSAR there are three layers: application software, RTE (Run-Time Environment) and the basic software.
- Operational states
Specification of system states and transitions between them (e.g. boot, shut down). In AUTOSAR some important common operational states are standardized.

After the architecture specification is finished (for a single ECU) the specification of a single software component need to be done. This includes the specification of a data model, a behavior model and a real-time model which are subsequently implemented in the design and implementation phase, [SZ10].

An exemplary model describing the software implementation architecture is depicted (in later sections) in Figure 3.4.

2.2.2. Architecture Description Languages and Standards

Due to the increasing complexity of software in automotive embedded systems (and the transition to new software development paradigm like MDD), there is a need to formally

²This is not a single possible solution, i.e. logical functions may be assigned to e.g. sensors, ADC etc.

2. Related Work

describe the software architecture. For this purpose, architecture description languages (ADL) are introduced, [vdBBFR03].

In the following sections several ADLs are introduced and compared for conformance to the automotive domain. Moreover, a very expressive comparison and classification methodology for ADLs can be found in [MT00].

2.2.2.1. AADL

The Architecture Analysis & Design Language is used for the formal description of hardware and software architectures in component-based development of complex embedded real-time systems. It is based on concepts of already existing ADLs (MetaH, Rapide, Wright, etc.), [FLV06].

AADL was firstly intended for the use in the avionic domain (Airbus, ESA) to support the development of safety critical embedded systems. In this time it was known as Avionics Architecture Description Language. From the user perspective, AADL specification provides a textual and a graphical notation of the language. The textual notation is a collection of declarations for components and their implementations, ports, packages and property sets. These constructs are used to accurately describe the application software running on an execution platform (hardware). In addition, an XML-based representation of the system is provided for model exchange purposes, [FGH06] and [FLV06].

In AADL three kinds of components are defined: (1) software components or application software, (2) execution platform and (3) composite or system components. Each software component has a predefined sub-program (executable code, e.g. libraries shared among processes), static data used by e.g. ports, process, thread group and threads. For task handling in a real-time environment AADL provides *rate monotonic* and *earliest deadline first* scheduling algorithms, [FGH06].

Hardware components or executable platforms contain a processor, a memory, a device (component interacting with an external environment) and a bus. The system is built by encapsulating bounded and instantiated components inside of composite.

In addition, AADL derives and extends the MDE concepts of the language MetaH. This allows to perform the analysis of the architecture (schedulability, safety, latency analysis etc.) and an automated integration such as runtime system configuration, application composition etc., [FLV06].

AADL is also realized as an UML profile and hence supported by several commercial and open source tools. For Topcased, an Eclipse plug-in Open Source AADL Tool Environment (OSATE) supporting the AADL metamodel is provided, [FLV06].

2.2.2.2. SysML

SysML is a modeling language developed by OMG (Object Management Group) to provide a more domain specific modeling than it is feasible with UML. It was built by reusing a part

2.2. Software Architecture for Automotive Systems

of an UML metamodel and extending it for domain specific constructs. These constructs address principally the system engineering domain including specifications, analysis, design and verification of complex (embedded) systems. SysML stands for Systems Modeling Language, [Hau06].

The specification of SysML describes the language in the following four basic diagrams, i.e. *pillars* of SysML: (1) structure, (2) behavior, (3) requirements and (4) parametric relationships.

The structure of SysML consists of *block* elements which are used for system composition. Elementary parts of the structure are encapsulated in *internal block diagrams* which are in addition parts of the system hierarchy described by *block definition diagrams*. These parts may further include blocks as parts, their ports (service-oriented and flow-oriented) and connectors. The whole system structure is organized in package diagrams, [Hau06].

For behavior modeling SysML provides constructs such as state machines, activities, interactions and use cases. They are modeled in separated behavioral diagrams, [OMG10].

Parametric diagrams in SysML are used to perform model analysis by describing constraints on system properties. These constraints may be mathematical expressions like $a = dv/dt$ performed on system parameters. They are encapsulated by the model element *ConstraintBlock*, [Hau06].

For data exchange purpose, two options are available. First, a model can be represented as XMI (XML Metadata Interchange) since the SysML profile is an extension of UML. Unfortunately, this realization is limited to model information only, i.e. no diagram information is included. Another option is the use of the AP233 (Application Protocol 233 based on ISO 10303 - a standard for data exchange) data exchange protocol, which is compatible to the SysML specification, [Hau06].

2.2.2.3. AUTOSAR

AUTomotive Open System ARchitecture - AUTOSAR represents an standardized and open automotive software architecture developed by more than 150 companies of automotive manufacturers and suppliers. The demand for such a standardization is caused by

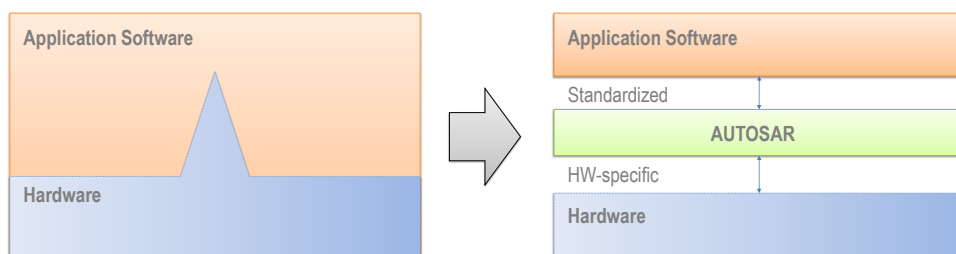


Figure 2.9.: Hardware/software interface: conventional (left) and AUTOSAR (right) [tre09]

2. Related Work

the growth of complexity in the automotive software development through the continuous increase of innovations in E/E systems. For instance, most of the automotive development processes in the last time have been resulted in ECU software that was highly dependent on the hardware. The reuse of such software would be a very time consuming task due to relocation of functions between different electronic control units (ECU). The solution provided by AUTOSAR to resolve these incompatibility issues is the introduction of an standardized layer between the hardware and the ECU software. This is illustrated in Figure 2.9. The purpose of this layer is to make the ECU software fully independent of the microcontroller and OEM³. Furthermore, this separation allows the efficient reuse of the application software and increases its flexibility [tre09].

Besides mentioned reasons for a standardized architecture, the automotive companies proposed a list of areas describing the main points where the standard should be applied. This includes implementation and standardization of basic functions, scalability across different vehicle types, embedding modules from other manufacturers, maintenance, upgrades and others [AUT08]. AUTOSAR response to this is a concept addressing the software architecture, the application interface and a methodology described in the following sections. The main goals of AUTOSAR is the achievement of higher software quality and reduction of costs for involved companies, i.e. OEM and suppliers. This should implicitly enable the development of more complex software as it is possible nowadays [KF09].

ECU Software Architecture An essential design concept of AUTOSAR is the separation between the ECU-specific and the ECU-independent software, i.e. basic software (BSW) and application software (ASW), respectively. Figure 2.10 shows the software architecture of the ECU in AUTOSAR. An intermediate layer called virtual function bus (VFB) acts

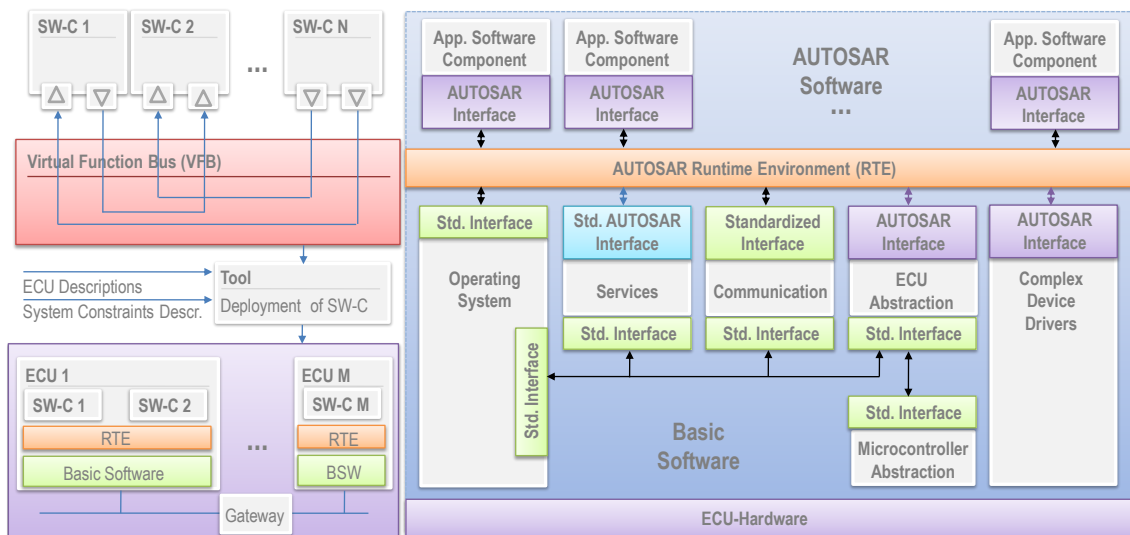


Figure 2.10.: AUTOSAR approach (left) and AUTOSAR ECU Software Architecture (right) [KF09]

³Original-Equipment-Manufacturer

as the abstract representation of the communication infrastructure for all software components belonging to the mentioned software concepts. Since its realization is ECU-specific, there is no information about the technology used. With such a virtual representation the independence between software components and the hardware is achieved and additionally the parts of an integration process can be done in an earlier design phase as before. From a VFB point of view that would be the connection of *Services*, *ECU Abstraction* and *Complex Device Drivers*. Since the communication construction is ECU independent, a high degree of modularity, scalability, exchange and reuse is achieved [KF09]. All this is located at the system level of the AUTOSAR methodology (see Figure 2.12). The implementation of the VFB is provided by the runtime environment (RTE) which is the core of the AUTOSAR architecture [AUT08].

Software Components In AUTOSAR software component represents a part of the functionality contained in the application running on the AUTOSAR infrastructure from which the component is fully independent. It consists of ports defined by (standardized) interfaces for communication purposes, a description forming the software component template (which for instance includes operations and data structures provided and required by a component) and the source code or ECU-specific object code. It is also important to mention, that each instance of the component can be assigned to only one ECU (*atomic* software components), but there is also a concept for *composition*, that allows to distribute the software components which are part of the composition over several ECUs [AUT08].

The inner parts of software components, i.e. the source code, is represented as *Runnable Entities* or *Runnables*. There can exist more than one *Runnable* inside a component. Each *Runnable* represents a piece of functionality provided by a component. For instance, a component containing two ports can be combined by two *Runnables*. The first one acts as the listener of the input port implementing the functionality to process the incoming data and saving it to some variable whose changes triggers⁴ the second *Runnable* which further prepares the output on the provider port [KF09].

Ports are used for interaction between components. They implement specific port interfaces which defines a type of the port and a data pattern for information exchange. It is also important to mention, that the same port types have different operational conditions in application software, calibration and AUTOSAR services. For instance, ports in application software are used for inter-application communication, whereas those for calibration are used for providing and consuming calibration values [KF09].

The AUTOSAR infrastructure tries to hide the specifics related to the microcontroller and the ECU electronics by using the abstractions of the microcontroller and the ECU respectively. An exception is made for sensors and actuators. They are represented by special software components that are independent from the ECU, but not from the sensors and actuators that are physically connected to the ECU. The reason for running such components on the ECU are performance issues [AUT08].

⁴they can be data or timer triggered

2. Related Work

Runtime Environment The RTE together with the basic software represents the technical realization of the VFB for a specific ECU acting as the middleware layer that handles the communication among application software components and their interconnection with the hardware over basic software by using the standardized interfaces [AUT08].

An essential prerequisite for the generation of the RTE is an XML based description containing the detailed specification of interfaces for communication. These information are embedded on ECU configuration description document, from which an extract specific for the current ECU is used for the generation of C-code. This code is in the further process finally linked with executables of software components [KF09].

Basic Software This is an standardized software layer providing the ECU functionality, i.e. access to the hardware for the application software. It consists of standardized and ECU-specific components. The first group includes *Services, Communication, Operating System* and *Microcontroller Abstraction* in which the second one includes *ECU Abstraction* and *Complex Device Drivers* [AUT08].

Considering the layered software architecture of AUTOSAR (the right part of Figure 2.10) the basic software forms the layered path to the hardware for system services, memory management, communication infrastructure and hardware I/O. The first level corresponds to the service layer, consisting of services for system, communication and memory. Communication services are responsible for the vehicle network communication (e.g. CAN, FlexRay), i.e. they provide an infrastructure for network management. Memory services are responsible for uniform provision of non volatile data (e.g. NVRAM) to the application as well as for a mechanism for their management. System services offer basic services like a real-time operating system and library functions that can be used by all mentioned modules [AUT08].

The next level in the layered architecture towards hardware is the ECU abstraction layer describing abstractions for I/O, communication, memory and on-board devices. An essential goal is to hide hardware details from software layers above. The first abstraction describes signals of on-chip or on-board I/O devices connected to the ECU excluding sensors and actuators. Communication hardware abstraction describes on-board and on-chip communication controllers and provides an uniform communication mechanism for them. In this way, it does not matter whether the controller is on-board or on-chip. The interface is the same. An analogous situation is present in the abstraction of memory hardware where the access to on-board and on-chip peripheral memory devices is provided with the same mechanism. The last abstraction describes on-board devices that are accessed by the ECU by using a microcontroller abstraction layer that abstracts I/O drivers, communication drivers, memory drivers and microcontroller drivers [AUT08]. The ECU abstraction layer with respect to the AUTOSAR methodology results in the configuration of the ECU and the generation of executables from the source code of software components [KF09].

There is also a part of basic software which does not belong to any layer of architecture named *Complex Device Drivers*. In this group the following modules are included [KF09]:

- modules that are not intended to be a part of the AUTOSAR architecture,
- modules with not realizable time constraints,
- modules from an existing project that are intended to migrate into the AUTOSAR architecture iteratively.

Communication Mechanisms Regarding the AUTOSAR methodology, this part belongs to the component layer, since it describes the communication patterns for software components. The first prerequisite for the realization of communication relationships is their modeling on the system layer (VFB). After modeling, the communication calls need to be implemented on the component layer. Additionally, the RTE should be provided and linked to the component implementation on the ECU layer [KF09].

Generally, there are two communication mechanisms used for component interaction: sender/receiver (*S/R*) and client/server (*C/S*) communication. The *S/R* principle allows to exchange data elements, i.e. primitive or complex data types, between sender and receiver. An important characteristic here is the transmission over separate communication channels. That means that the sender doesn't get an acknowledge on the same channel after the transmission. Instead, an additional channel for acknowledges is required. This allows the efficient use of multiplicity in communication design. It is possible to realize communication in such a way, that the sender performs multicasting, i.e. sends its data to more than one receiver and vice versa. Besides mentioned features an important characteristic of *S/R* communication mechanism is the explicit and implicit communication. The first form corresponds to the direct establishment of communication by using the appropriate API. The receiver is equipped with a data buffer, i.e. queue with constant length. The second form, i.e. implicit communication allows to exchange data between components by using the RTE as an intermediate layer. A specific feature in this communication form is the transmission start after sender termination. On the other side, the RTE transmits data to the receiver at the time when he is ready (active). In this way, transmission data will not be send immediately. Instead, they will be prepared for transmission during the runtime of the sender. From the receiver point of view, the advantage is that he can request data from the RTE at any time without having a copy of the data in his local space.

The *C/S* communication mechanism in contrast to the *R/S* provides bidirectional transmission. Instead of the sender and receiver, here client and server are related in a $1:n$ communication relationship. This means, the server port, i.e. the operation inside of this port, can be used by more than one client. The reverse direction is not allowed [KF09].

The generation of the RTE results in different APIs for the mentioned (different) communication principles. An essential feature is that the API definition does not contain information about source or target components (e.g. sender or receiver), but only their ports defined in earlier development phase on the system level (VFB). This allows to fulfill one of the most important goals of AUTOSAR, namely, the exchange and movement of software components [KF09].

Figure 2.11 shows an exemplary warning lights system described on system level in order

2. Related Work

to demonstrate various communication mechanisms altogether. The main element here is the *Warn Light Application* which is already a composition of two software components. It has two inputs: *Velocity Calibration* from the calibration component and *Warn Button* from the sensor software component. Here it is clear that the *S/R* communication principle is used, but it is also noticeable that it is used for different application areas, namely, calibration and application software components. The *Warn Light Application* performs the multicast to the actuator components *Warn Light Left* and *Warn Light Right*. Here the *S/R* principle is used too, but only for application software components. Furthermore, the actuators are connected with the light outputs of the ECU by using the *C/S* communication principle, because they need to send commands only (turn off/on). Another interaction of the *Warn Light Application* is realized with the basic software component *ECU State Manager* over *Mode Manager*. Thus, for retrieval of current ECU status the *S/R* principle is used and for the *RunMode* the *C/S* principle is used. Both communication paths belong to the AUTOSAR-service area.

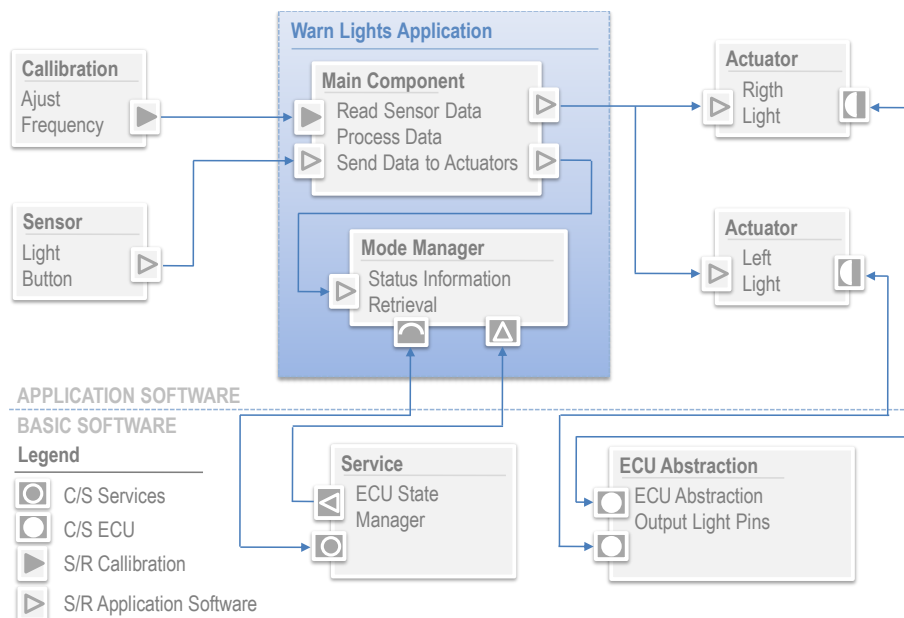


Figure 2.11.: Example: warn lights system realized in AUTOSAR, [KF09]

Methodology The AUTOSAR methodology describes the steps, i.e. work flow for developing AUTOSAR products. In this set of activities three abstraction layers are considered: system layer, ECU layer and component layer. They are also related to a conventional system development process which is here partly automated. Figure 2.12 shows a basic work flow describing the creation of an AUTOSAR project from the given system description. Since the output of each activity is a standardized XML file format, the whole process can be supported by a tool. Additionally, the exchange of any part of the project between different tools conforming to AUTOSAR is possible [KF09].

The main starting point of the work flow on the system level are the resources *System Con-*

figuration and *Collection of Software Component Implementations* provided by the OEM. The system configuration contains information about the allocation of software components to ECUs. Additionally, it can be specified which software implementations are used. Both documents are inputs for the *Configure System* activity which results in resources for further finer grainer configuration activities (i.e. *System Configuration Description* and *System Communication Matrix*). The first resource is required in the next lower layer of the AUTOSAR methodology to extract relevant data from several ECU descriptions, whereas the second one contains the information flow of the whole system [KF09].

Extraction of relevant data from the *System Configuration Description* is done by the ECU supplier, because very broad knowledge about the target system is required in this step. The ECU supplier is responsible for the configuration of the basic software, the RTE and the operating system for his ECU. The product of this activity is a resource called *ECU Configuration Description*, which in combination with the software component implementation allows to generate an executable for a given ECU (see ECU layer in Figure 2.12). Finally, the last abstraction layer of the AUTOSAR methodology includes

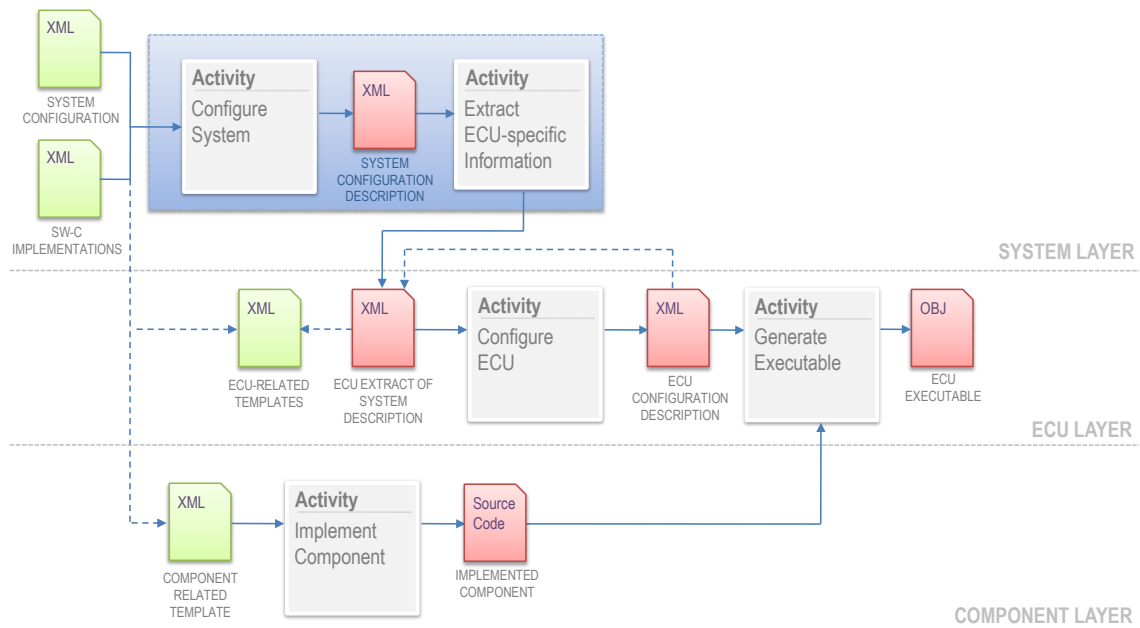


Figure 2.12.: Development of ECU software based on AUTOSAR methodology, [KF09]

the component implementation flow. As mentioned in the previous sections, AUTOSAR components can be developed without being aware of the target hardware. Therefore, the activity *Implement Component* is equal to the software development process which results in source code of the software component which is later compiled for a specific ECU [KF09].

Application Interfaces Since the functionality of AUTOSAR software is encapsulated in interacting software components, an essential feature towards better scalability, exchange and integration of such functionality is the standardization of communication interfaces.

2. Related Work

In the AUTOSAR architecture, described in Figure 2.10, three standardization forms are used: (1) AUTOSAR Interface, (2) Standardized AUTOSAR Interface and (3) Standardized Interface. The last group of interfaces is standardized by an external party (e.g. C programming language) [AUT08].

Exchange Data Format For application independent data exchange AUTOSAR provides an XML schema which conforms to its metamodel, extended by variability modeling means. The XML schema is developed with respect to the ASAM FSX (Association for Standardisation of Automation and Measuring Systems) - a function specification exchange format.

Variation Management in AUTOSAR Documenting variability in AUTOSAR is realized by variation points, binding time and binding expressions. The last one corresponds to an expression definition which is used to trace variable elements to configuration parameters, i.e. to system constants. The AUTOSAR variant management mechanism covers the following binding times:

- System design time
 - Corresponds to the early phase in system development, e.g. designing the VFB, specification of software component types, building connections between prototypes, etc.
- Code generation time
 - Writing or generating (or both) the source code.
- Pre-compile time
 - At this time an object code is generated by combining the parts of the code according to selected variants. This selection is performed by a preprocessor.
- Link time
 - Corresponds to the configuration of the resulting object code by excluding/including its parts (modules).
- Post-build time
 - This is the latest binding time in AUTOSAR. It covers the binding of variants between generated executables and operation time of the ECU. For instance, this includes flashing different configurations into the RAM of the ECU before its startup.

The first four binding times correspond to the pre-build time. Thus, function design time and runtime are out of scope for this mechanism. In earlier versions (before version 4) of AUTOSAR the variability is not explicitly defined. Instead, configuration steps from the AUTOSAR methodology are typically used as possible candidates for handling different variants externally.

The support for the variant management in AUTOSAR is not an integral part of the

AUTOSAR *core* metamodel. Instead, an extended metamodel is generated to provide the variant management. The XML schema is derived from this extended metamodel. In following, the generation of the AUTOSAR XML schema is explained (see Figure 2.13).

The AUTOSAR XML schema is generated from the *Pure Metamodel* by applying metamodel patterns to certain parameters. This strategy allows to keep the metamodel highly flexible. The generation starts by annotating the *Pure Metamodel* with additional data. Here it can be specified which elements of the metamodel are intended to be variable by extending elements for attributes like the latest binding time. These annotations are crucial input for the schema generator. Namely, it takes the *Annotated Metamodel* and in correspondence to annotations and defined patterns it generates the *Extended Metamodel*. There are now all elements to support the variant management. Another reason for

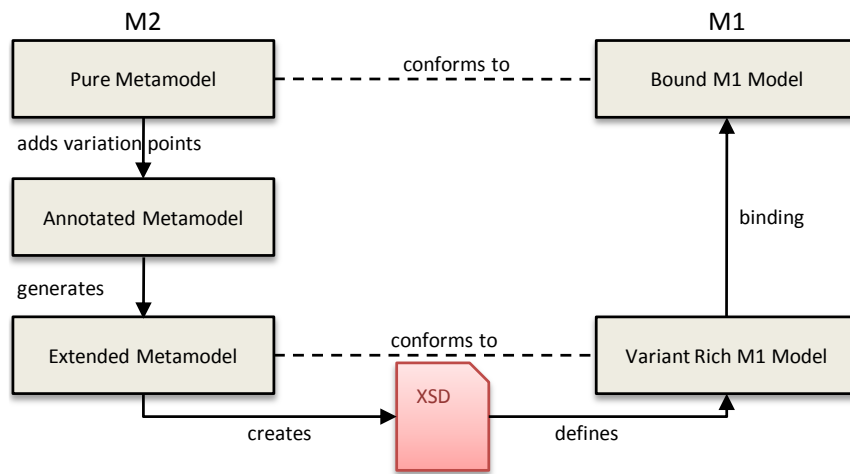


Figure 2.13.: AUTOSAR XSD generator, [AUT09a]

this extension are model constraints. A model generated by resolving all system variants is a variability-free model that should conform to a variability-free metamodel, because constraints regarding e.g. multiplicity may be not the same. For instance, to allow client-server communication where the client in variant 1 is connected to server 1 and in variant 2 to the server 2, the upper bound of multiplicity between requester and provider port need to be changed from “1” to “*”. Thus, in a resolved model this multiplicity need to be restored. From the *Extended Metamodel* the AUTOSAR XML schema is generated.

An excerpt of the extension from the *Extended Metamodel* are locations for variation points. They are aggregations, associations, attribute values and property sets. The XML schema generator uses the predefined patterns to create this excerpt in the XML schema. These patterns are:

- Aggregation pattern
- Association pattern
- Attribute pattern

2. Related Work

- Aproperty set pattern

A variation point inside an aggregation is used to determine if a part aggregated to a container exists in the system and under which circumstances it exists. An example for such a variation point is a port inside a software component. The binding time for this variation point is explicitly provided, but also constrained by the latest binding time. This latest binding time is introduced because some variation points are not allowed to be bound at a specified binding time (cf. attribute value pattern).

The situation with the association pattern is almost the same. A variation point is aggregated to the reference object building the conditional reference. This allows to make references variable. However, this is not the case with all references (e.g. referencing types).

A pattern applied to variable attributes differs from the other two mentioned before in a way that the existence of an attribute is not configurable, but its value. Here, a set of data types that can show a variable character is defined. It is enough to extend these attributes with a binding time to make them configurable. The latest time to bind these variants is the pre-compile time.

If a large number of attributes is required, the usage of the attribute pattern makes the situation difficult, because for each attribute the variation point needs to be specified individually. Moreover, the attribute pattern data type has restrictions. Finally, there is no way to handle such attributes if they are dependent on each other, i.e. if value sets are assigned to them. To solve this, a property set pattern is provided. It encapsulates one set of values for all attributes in a varying container. In this way, a set of values for a large number of attributes can be easily configured. Additionally, to provide variable values of value sets the attribute pattern can be used.

Variation Point As previously mentioned, a variation point describes variable locations in a model. But it is also important to know how it defines conditions under which such locations exist. This excludes the attribute pattern, because for its realization another kind of variation point is used, namely the attribute value variation point.

A variation point aggregates elements describing a condition by formula and a post build condition. Both conditions are used to express a criterion that specifies the existence of the underlying variant. The first is used for variants intended to be bound before post-build time, whereas the another condition is related to post-build time. They can also be combined into a single variation point. In this case there are 16 possible resolutions of variants defined in [AUT09a].

The syntax of the formula condition is similar to the C programming language and its grammar is realized in ANTLR⁵. A single variable inside of this expression is a system constant. Its values are specified by value sets before any binding time. These value sets are in the form of a table, which maps system constants with their values. Actually, this

⁵ANother Tool for Language Recognition - parser generator

table represents the approved product line for the underlying model. If a formula condition results in a 0 this variant needs to be removed from the model. Otherwise, it is included until the next binding time defined by another condition, if any.

A condition for post-build variants is not defined by formula, but it works in a similar way. It contains a value that need to be matched with criterion. If more than one conditions is specified, all of them need to return true in order to bind the variant.

From the VFB perspective, [AUT09d] defines three metamodel elements that may show variable character: software components, ports and connectors.

Component Reuse AUTOSAR follows the UML strategy *type - role* to reuse model elements. Actually, in AUTOSAR a role is defined as a prototype. In this way, types of e.g. software components may be instanced multiple times. Moreover, this reduces the redundancy in modeling, because prototypes are not-decomposable black-box elements without any type-specific attributes (except of a reference to a type).

2.2.2.4. Fibex

Fibex is a XML-based, standardized exchange format describing the communication among different bus systems in a vehicle. It is developed by ASAM as an alternative to CAN (Controller Area Network) and LIN (Local Interconnect Network) files and stands for Field Bus Exchange Format, [ZS07].

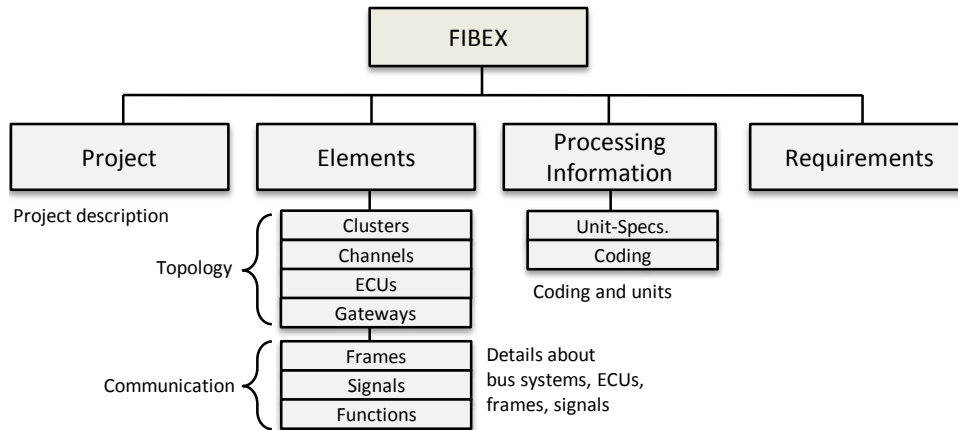


Figure 2.14.: FIBEX XML schema structure, [ZS07]

Bus systems in a vehicle can be categorized into systems for on-board and off-board communication. The first group includes low speed, high speed and multimedia systems exchanging data between ECUs, whereas the other systems are intended for the communication to the external world (e.g. for diagnostics). Fibex is focused on on-board communication systems. The structure of its specification is depicted in Figure 2.14.

The main block in the structure is called *elements*. It holds on the one hand the topology

2. Related Work

consisting of bus systems (*clusters* with one or more *channels*) and ECUs (incl. *gateways*) and on the other hand the *frames* and their content (*signals*).

The currently supported bus technologies are FlexRay, MOST (Media Oriented Systems Transport), CAN, TTCAN (time triggered) and LIN, [Bar09].

2.2.2.5. EAST-ADL

Electronics Architecture and Software Technology - Architecture Description Language is, as the title says, a language for describing architectures in the automotive domain. It is developed in the scope of the following European research projects: the EAST-EEA (EAST - Embedded Electronic Architecture) between 2001 and 2004, then ATESSST and ATTEST2 (Advancing Traffic Efficiency and Safety through Software Technology) between 2006 and 2010 and finally its further development should be completed until 2013 in the scope of the project MAENAD (Model-based Analysis & Engineering of Novel Architectures for Dependable Electric Vehicles).

Its primary goal is to provide a detailed documentation of an integrated system and to improve the communication in the development environment. This is first of all achieved through a representation of a system in different layers of abstraction and additional modeling aspects like variability, requirements modeling, feature modeling, environment modeling, system level analysis, etc., [CFJ⁺10].

The EAST-ADL metamodel is provided as an UML profile (an extension of the UML to provide domain specific modeling construct, i.e. a domain specific language) which is like UML on M2 level in the MOF metamodel hierarchy [OMG07].

Structure As previously mentioned, EAST-ADL distributes the engineering information over five abstraction layers. This is illustrated in Figure 2.15.

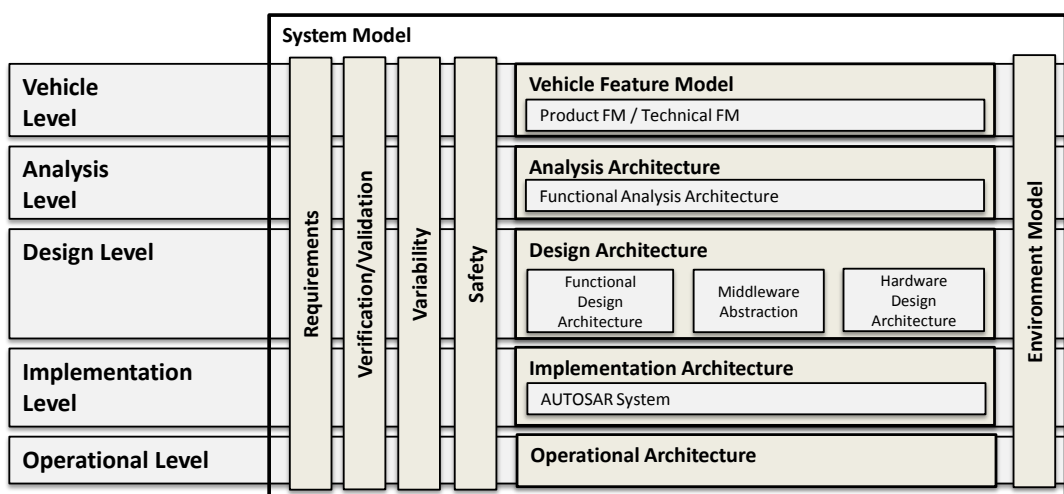


Figure 2.15.: EAST-ADL domain structure, [CFJ⁺10]

The vehicle level is used to provide an abstract description of the whole system in form of features. This allows to establish a communication to a stakeholder (e.g. customer) in order to manage the underlying product line (configure the system, modify the PL mechanism, etc.). However, this level does not describe how the product line is built internally. Instead, it just describes what the system has (for PL in EAST-ADL see the paragraph *Variant Management*).

On the analysis level abstract functions are defined by decomposing requirements and features. This abstract functional description corresponds to a domain concept i.e. the environment model, devices interacting with an environment and functions, [CFJ⁺08].

In the next level devices are further decomposed to either hardware (e.g. sensors, actuators) or software elements (for signal transformation; they are not an application software). Middleware is modeled additionally to connect device-specific functions to design functions. These functions form the *Functional Design Architecture* (FDA). Parallel to this, the *Hardware Design Architecture* (HDA) encapsulates an abstract description of the hardware.

The implementation level is not explicitly defined in the EAST-ADL metamodel, but instead, an AUTOSAR implementation architecture is used. The reason for this is an extensive support for realizing the implementation architecture in AUTOSAR (e.g. basic software, detailed software topology, etc., [CFJ⁺08]).

The operational level corresponds to the system installed into a vehicle.

Variant Management The vehicle level model is the main point for variant management in EAST-ADL. It is combined from optional and mandatory features in such a way, that very abstract description of the whole system can be captured. This allows to define what the system has from a variability point of view, but not how it reacts on this variability. The reaction is defined in lower levels of abstraction, where the variable parts, i.e artefacts, are represented as variation points. In order to instantiate the variable-free product, the relation between the artefact level variability and the feature model on the highest level of abstraction must be specified. Finally, the product derivation is driven by the feature model on the highest abstraction level. Basically, there are two aspects of variability in EAST-ADL:

- Vehicle feature level variability
 - Core technical feature model
 - Feature model for end-customer configuration
 - Feature model for non-customer configuration
- Artefact level variability

The first variant of variability representation belongs to the highest level of abstraction. It results in a feature model combined from optional and mandatory features that are used

2. Related Work

on the one side for variant management activities and on the other side for a coarse-grained specification of requirements. The distinction between mandatory and optional features is expressed through cardinality, [ea08].

Except of the capability to express the features as optional and mandatory, i.e. $[0..1]$ and $[1]$ respectively, it is also possible to clone those features, i.e. multiplicity > 1 . As a consequence, multiple instances of the same feature with different or the same configurations may be created. For instance, a product derived from the product line has the front and the rear wiper system with the rain control functionality including timing intervals for the pause between the activation of the motor. In the product line architecture, there is no need to specify the features explicitly for each wiper system, but only one with cardinality 2. Later in the configuration step, two instances of such systems are created. The effect of this configuration is part of artefact level variability.

Vehicle Feature Level Variability The features from the vehicle feature model (in the next VFM) address different feature groups as well as different models. Some features describe more technical aspects than others (e.g. a brake system in contrast to the market) and some of them are probably visible for the customer. Therefore, EAST-ADL separates them on the one side in technical and non-technical feature models and on the other side in customer visible and customer invisible feature models.

Core Technical Feature Model Features from this model reflect the pure configuration of the system, i.e. no characteristics like market, customer-visible variants, etc. are present here. Such categorization of feature models reduces complexity of the system configuration, because this model is typically a product of some pre-configuration and hence contains less variable domain artefacts.

Feature Model for End-Customer Configuration The core technical model is not visible for the customer, only for the system engineer. However, a customer may require to choose some variants by himself. In EAST-ADL this functionality is provided by configuring the separated feature model, i.e. the product model. The mapping between both models is done by the product decision model depicted in Figure 2.16.

Feature Model for Non-Customer Configuration This model in contrast to the previous ones has internal features only. Thus, they are also not technical features. An example of such feature models is a market containing different countries as features. Like in customer-visible configuration this model is used to pre-configure the core technical model in order to reduce the complexity of the product line.

Product Decision Model The core technical model serves as the basis for the configuration of the whole system. It may be pre-configured by two other vehicle level feature models. The relation in the pre-configuration process is known as configuration link or decision model (see Figure 2.16).

The product decision model is a simple list of decisions which for a given source model

and given rules configures a target feature model. This strategy is used to pre-configure the core technical model. In the example above, the functions $F1 - F4$ are a part of the source model. Their exclusion and inclusion criterion defines the mapping, i.e. what is a consequence to their presence or absence in the source model. In addition, more complex expressions in the selection criterion may be defined. This expression corresponds to the grammar of VSL (Variability Specification Language).

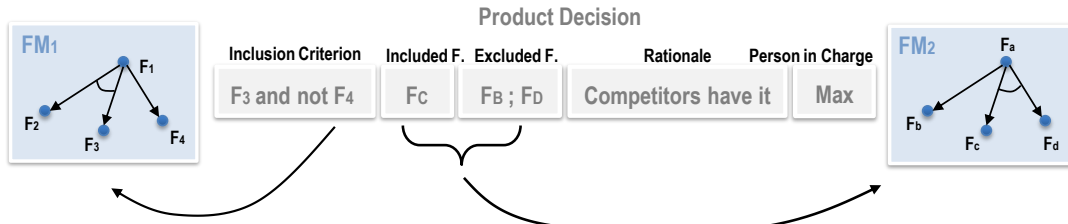


Figure 2.16.: Product decision model, [ea08]

The way how EAST-ADL configures the system is defined as follows: on the highest level of abstraction, the core technical model is configured. An effect on this configuration is realized in decision models which can be assumed as traces to model artefacts. The derivation process evaluates the selected features with respect to these decision models and creates a new variability-free model.

Artifact Level Variability Variant management on artifact level relies on public feature models and internal bindings. They are the most important metamodel elements for fine grained system configuration. They trace the configuration from the vehicle level to analysis, design and implementation variabilities. These variabilities are described in an OVM provided as an extension in EAST-ADL.

Artifact Level Feature Models The essential characteristic of the artifact level variability is the presence of feature models on lower levels of abstraction. In other words, there are additional feature models describing domain artifacts more accurately than in vehicle level. This is illustrated in Figure 2.17.

The composite design function $F1$ contains one elementary and one composite function, Fa and Fb respectively. The function Fb is further decomposed in a similar way. Now, in order to handle such architecture-centric variability (see Section 2.1.1.6), each composite function gets its own feature model (public feature model - PFM). The purpose of this model is to configure the content of the underlying function only. However, a composite function may contain other composite functions which also have their own public feature models. In this case the feature model of their parent function is used to configure these public feature models instead of configuring the content only. In the example above, this would mean that the PFM of $F1$ configures the PFM of Fb (but also the elementary function Fa), which in addition configures the PFM of $Fb2$. Finally, this PFM has to configure the content of the function $Fb2$, i.e. elementary functions $Fb2a$ and $Fb2b$. This configuration propagation allows to refine the variability in lower levels of the hierarchy.

2. Related Work

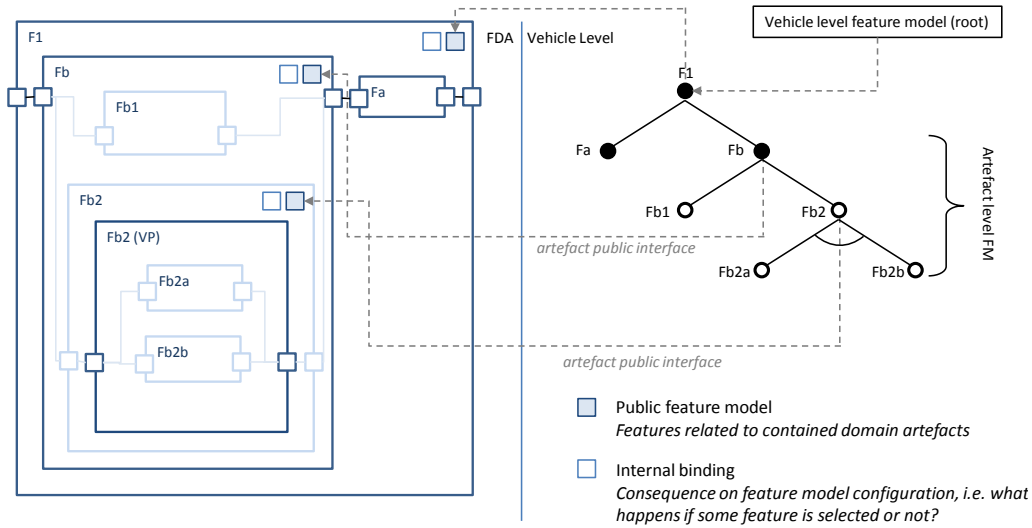


Figure 2.17.: Bridge between two levels of variability in EAST-ADL

Compositional Variability As previously elaborated a particular public feature model in combination with an internal binding may configure elementary and composite functions. For the second group, a configuration is usually related to the PFM of contained functions. This kind of variability managed in a hierarchical fashion is known as *compositional variability*, [RKW09]. It exploits issues on application of traditional product line techniques applied in automotive (see Section 2.1.2.1) to provide a high flexible fine grained variant management for component-based design. An example related to this topic is given later (see Figure 3.5).

Figure 2.17 illustrates the compositional variability by structuring features in *reference - referred* feature model relation. The reference model in this context is the one from the upper level from which the referred model is derived. Its specialization inherits the parent features with respect to predefined rules, i.e. a *deviation set*. The whole product line feature model, i.e. VFM and all PFMs, forms a multi-level feature tree which enables the independent development of parts of the system, i.e. *product sublines*. This is very important for the OEM-supplier relation, because both parties can easily agree on the interface between their product lines.

A high degree of flexibility is achieved by providing different realizations of a configuration link. This is in [RKW09] documented by the following patterns:

- Plain propagation - one-to-one mapping (configuration is propagated to the parent FM in upper level).
- Direct binding - configuration of certain variability is invariant, i.e. a constant value constrained by its container.
- Orthogonal propagation - configuration propagation by additional feature models (e.g. for limitation of possible values).

- Top-level propagation - direct linking between lower level variability and top level feature model (actually contradiction to compositional variability, but in some cases it is required).
- Reverse propagation - configuration by using global features (cf. AUTOSAR variant handling mechanism, Section 2.2.2.3).

The main benefits of compositional variability are the reduction of complexity while building the platform due to the high degree of flexibility and a provision on different views of the system through interrelated multi-level feature models.

Binding Times EAST-ADL covers AUTOSAR binding times (system design, code generation, pre-compile, link and post-build) and in addition allows to configure the system at runtime.

2.3. Model Transformation

The need for model transformation has its origin in the lack of a technology supporting the synchronization and mapping between models, generation of lower-level models and source code, reverse engineering of higher-level models, customizing model views, etc., [CH06]. Actually, the demand on such a technology was an attempt of the OMG (Object Management Group) to provide a transformation between the PIM (Platform Independent Model) and the PSM (Platform Specific Model) in MDA (Model Driven Architecture), an model-based software development approach in MDE [CH03]. In 2005 the OMG has published QVT (Query/View/Transformation), a specification which in three transformation languages, i.e. relations, a core and an operational mapping, describes the transformation. The first two languages provide a declarative way to map the models (e.g. by using Object Constraint Language - OCL queries to access the model), whereas the operational mapping is an imperative language using an appropriate API in a transformation process to access the model, [CH06].

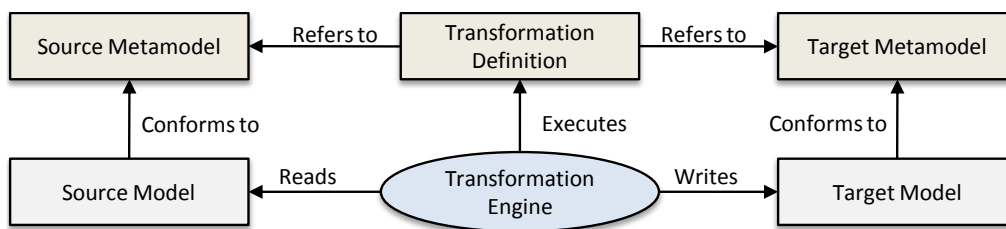


Figure 2.18.: Concept of model transformation, [CH06]

Figure 2.18 shows the concept of the model transformation. The transformation engine generates the target model from the source model with respect to a predefined transformation definition containing the mapping rules. Both models have to conform to their metamodels, which are not obligatory different to each other (to provide e.g. a mapping between different abstraction layers of the architecture described by a single metamodel).

2. Related Work

Moreover, the transformation allows to include multiple source and target models, [CH06].

Transformation rules specify the detailed procedure of the mapping by providing variables, patterns and logic for involved models separately. Variables correspond to model elements (meta-elements) being involved in a transformation. Patterns are a container for variables in form of expressions, terms or a graph, whereby the logic, which can be declarative or imperative, describes the computation and constraints between model elements, [CH03].

2.3.1. Model Transformation Approaches

In [CH03] different transformation approaches are classified into two categories: model-to-code (or text) and model-to-model transformations. Some of the approaches are essential for the design decisions in the practical part of this thesis.

2.3.1.1. Model-to-Text Transformation

In model-to-text (or code) transformation, two basic approaches are present: a *visitor-based approach* and a *template-based approach*. The strategy of the first transformation is to traverse the model by using a visitor which generates the source code in correspondence to the visited elements and a given transformation definition. Unlike this, the other approach uses templates to generate the source code. The part for processing the source model (known as *lefthand side* (LHS) in [CH03]) is considered to be an access point for the source model (e.g. an API written in Java, or declarative XPath, etc.) which in combination with the string pattern containing a metacode (rules for selection of the code from a source model) is used by the *righthand side* (RHS) part of the transformation to generate the target source code. This solution is more flexible, because the rule specification does not bind the transformation to a specific language.

2.3.1.2. Model-to-Model Transformation

Direct Manipulation Approach This model-to-model approach relies on a custom built rule definition which uses an appropriate API as the transformation engine (see Figure 2.18). On the one side there is functionality to manipulate both models which is bridged by a mapping API and on the other side there is an *empty* transformation definition which must be realized for specific needs.

Relational Approach The relational approach is based on mathematical relations between a source and a target model by using constraints (e.g. QVT relations language, [CH06]). Such transformation definitions are declarative and hence have no executable character. They are typically driven by a transformation process. In such a way a high flexibility is reached. Furthermore, such an approach supports the bi-directionality in model transformation.

Graph-Transformation Approach This approach is based on the graph transformation methodology. Its LHS part of the transformation is a graph pattern reflecting the source model, whereas the RHS graph pattern describes the target model. These patterns are

specified either by a concrete or by a MOF based abstract syntax. The second is more common, because it allows to express the transformation without being fixed to a specific metamodel.

Structure-driven Approach The structure-driven approach requires the transformation rules (definition) only, since it provides a framework for a model transformation. The background of the process bases on building an internal hierarchical structure of the target model and subsequently building references between involved models. In correspondence to a rule definition (a function of source and target types in a predefined form) model elements from the source can be simply *copied* to the target (cf. OptimaJ framework in [CH03]).

Besides the mentioned approaches, some of the existing transformations are in [CH03] assigned to a hybrid group, which is a mixture of different transformation definitions from both categories (e.g. ATL - Atlas Transformation Language combines a declarative and an imperative transformation strategy, [CH03]).

Further very helpful information concerning the implementation and the feature-based customizing of the introduced approaches can be found in [CH03] as well as in [CH06].

2.4. Tool and Language Evaluation

2.4.1. ADL Selection and Evaluation

ADL evaluation					
Supported Feature		AADL	EAST-ADL	SysML	UML
Function modeling					
	Components	✓	✓	✓	✓
	Ports	✓	✓	✓	✓
	Signals	✓	✓	✓	
Hardware modeling					
	ECU nodes	✓	✓	✓	
	Buses	✓	✓	✓	
	Sensors/Actuators	✓	✓	✓	
Environment modeling			✓		
Requirements modeling			✓	✓	
Safety analysis		✓	✓		
Variant management			✓		
Rich tool support		✓		✓	✓
Automotive standard					
Automotive domain summary [%]		66,67	83,33	66,67	25,00

Table 2.2.: Evaluation results for architecture description languages for the automotive domain

2. *Related Work*

This section describes a short evaluation of architecture description languages (ADL) for a given set of features that need to be supported. These features are collected by the HybConS team internally. Table 2.2 shows the results of the evaluation. The last parameter “Automotive domain summary” shows the summary information expressed in percents. It corresponds to the coverage of supported features. The aim of this evaluation is to find the best suited ADL for modeling embedded software and hardware in the automotive domain. It is important to mention that AUTOSAR is excluded from the evaluation, since it is more a methodology than a language. But anyway, it is also possible to use the UML profile of AUTOSAR to describe the implementation architecture of the system.

The result shows, that EAST-ADL is suited best to describe the automotive domain. Unfortunately, it has a very poor tool support and it is still not widely used in the automotive industry, but this could change in the future. However, its detailed description of the engineering information is satisfying for the purposes of the HybConS project.

SysML and AADL are not so different from EAST-ADL, but they are not “enough” domain specific. For instance, SysML supports the definition of ports and components, but it is a common description. In EAST-ADL this description is further specialized to address the automotive domain more precisely. The situation with AADL is the same. In addition, the system is focused on the implementation level only. Thus, one of the most important structural characteristics of EAST-ADL is the distribution of the engineering information over several abstraction levels and the ability to make traces between these levels. Its aim is to document the whole integrated system.

2.4.2. Tool Selection and Evaluation

2.4.2.1. Selection Criteria and Prerequisites

The task of tool evaluation and selection for product line engineering is not trivial. Especially in case of additional prerequisites which need to be respected. Typically, the task is performed through prioritization of selection criterion derived from best practices described in Section 2.4.2.2. This prioritization is domain specific and therefore requires deeper analysis which doesn't result without some risks related to the product line [BCD⁺00].

The first evaluation phase resulted in EAST-ADL as appropriate architecture description language for the representation of the product line architecture of automotive embedded systems. This result has big impact on the tool evaluation and selection. It reduces the range of PL, DSL or MDA tools to those supporting the EAST-ADL profile and its structural constructs (e.g. variability, feature modeling, hardware modeling, requirements, etc.) or at least to those tools having a high capability to create a meta-model forming a convenient domain model and having high extensibility. In short, the target tool must support the EAST-ADL domain model (meta-model) in any way.

To provide systematic reuse of core assets, their management and creation and on the other side to provide a (partly-) automated process for product derivation, the essential characteristic of a tool is the support for both, domain and application engineering (in own as Application to Core Asset Development and Application to Product Development respectively). This involves tools for configuration management, requirements discovery and management, architecture modeling, reverse engineering, impact analysis, regression testing, project management, economic modeling, design modeling and build management [BCD⁺00]. It is also not enough to have a set of tools providing the above mentioned features individually. Each value in criteria for tool selection includes a group of features that must be satisfied by a tool (e.g. validation is applied in tool providing reverse engineering, regression testing and impact analysis [BCD⁺00]). Therefore, as a result of selection an integrated tool environment supporting the product line is imaginable. This implies additional interoperability issues which are one of the most important challenges in further development.

The essential characteristic of a tool for application engineering (or product engineering, see [ea08]) is product derivation. Typically, the product derivation process takes the specification and a set of rules and generates a product. There are also differences between tools regarding the degree of automation. Some of the tools support automated product derivation, whereas others support it only partly. The main problem with partial product derivation is the integration of a set of tools, which typically lack of interoperability. It is also difficult to follow the way of derivation from the domain containing the product line artifacts down to the generated product artifacts containing executable code, unit tests, architecture documentation, etc. To satisfy the requirements of the project described in this thesis, more priority has been given to tools supporting the longest derivation path or ideally a tool supporting the whole path to completely avoid these interoperability issues. Alternatively, it is also imaginable to choose a tool without automated product derivation,

2. Related Work

but with high extensibility.

The meta model of EAST-ADL contains constructs for feature and variability modeling. For variability modeling, orthogonal variability models are used to describe variability on lower levels of abstraction. They represent the fine grained structure. A feature model, in contrast, describes the system in a more abstract way. Without these modeling aspects it is neither possible to form the product line architecture nor to instantiate the products. Therefore, tool support is mandatory.

Engineering information in EAST-ADL is distributed over five different abstraction layers. To keep information consistent and to react on changes in a model, information traces between and inside the core assets are required. EAST-ADL includes these traceability feature in its language definition. They should be supported by a tool as well. A critical point regarding traceability is the binding between the feature model on the vehicle level and the variability models on artifact level (FDA, HA and IA, see Report 2). This binding is a prerequisite for product derivation from the configuration instanced from the core technical feature model on the vehicle level. If this binding is not supported by a tool, it must be implemented if the extensibility capabilities allow it.

One of the most challenging tasks in this project is to find a way to transform the Simulink® models into a structured product line domain model. Ideally, this process can be automated. There are some remaining issues, for example, how to resolve commonalities and variabilites from a repository with respect to some specification and how to distribute them to form the product line architecture with traceability links to corresponding requirements. In this area, model comparison is an essential feature that should be supported by a tool. The next important step is validation checking to ensure that a generated model corresponds to the metamodel. A positive validation is a prerequisite for successful product derivation.

2.4.2.2. Evaluation Methodology and Tools

Allocating the appropriate tool for the product line support is done based on activities described in [Sit]: identification of needs, measurement, evaluation and selection. The first activity describes the requirements to be supported by a chosen tool or tool chain. Such a list of requirements helps to reduce the number of tools which have to be actually evaluated. The measurement represents the engagement of a tool in the practice in order to compare it with previous experiences [Sit]. Evaluation activity is similar to measurement. It acts as a basis for tool selection providing enough information to compare the tools (see sections below). The last activity is the most important one. It is based on a weighted list of criteria taken from the master thesis' of Andrea Leitner [Lei09] and Andreas Haselsberger [Has10] and is extended by some additional attributes based on [BCD⁺00], [ODF07], [DDN07] and [Sit].

The tool environment of EAST-ADL is separated into a modeling workbench and an analysis platform. The analysis platform corresponds to plugins needed for e.g. validation check, product derivation, model tranformations, etc.. The modeling workbench is the

base modeling tool representing the EAST-ADL model graphically and semantically. For successful tool evaluation and selection it is important to be aware of mentioned aspects, since the resulting tool or a tool environment needs to be aligned with the concepts of the analysis platform and a modeling workbench.

In the following sections the tools, included in the evaluation and selection process, are described and the results of the evaluation are presented. These tools are: Papyrus, MetaEdit+, pure::variants, openArchitectureWare, Enterprise Architect, MagicDraw UML and a prototype of the Common Variability Language (CVL). First it is important to mention, that the tools are belonging to different areas of software development and methodologies. MetaEdit+ belongs to the category of DSL tools (Domain Specific Languages), openArchitectureWare is a MDA-based (Model Driven Architecture) tool, Enterprise Architect and MagicDraw UML are tools for general purpose modeling and finally pure::variants and CVL are tools for Product Line Engineering (PLE).

Papyrus Papyrus is an open source modeling tool which has been developed in the scope of the ATESSST project by CEA LIST⁶ and acts as the modeling workbench for EAST-ADL. Basically, it corresponds to a customized Eclipse modeling platform to satisfy and provide modeling concepts addressing embedded systems (as well as those for the automotive domain). Therefore, other UMLTM profiles like SysML, MARTE and CCL can be applied instead of EAST-ADL [Pap]. It is now a part of EMF (Eclipse Modeling Framework). In the following text, Papyrus will be used as synonym for Papyrus extensions for EAST-ADL.

Papyrus can be used as a plugin in the Eclipse modeling platform or alternatively as a standalone distribution [Pap]. As it has been developed in the scope of the ATESSST project, the consistency between the EAST-ADL metamodel and supported metamodel is always up to date. This avoids interoperability issues and the need for an explicit update of the metamodel. In other words, a tool evaluates with a metamodel together.

Modeling Workbench The support for EAST-ADL in Papyrus is just a specific purpose of a tool, i.e. it's configuration. There are two important characteristics to assume: modeling means (e.g. diagrams) and UMLTM profile for EAST-ADL. The first one is realized by the EAST-ADL customization plug-in based on EMF. It supports the creation of stereotyped UMLTM constructs corresponding to the EAST-ADL profile directly in the model [SN10].

The transformation of the EAST-ADL design architecture to AUTOSAR software components is illustrated in Figure 2.19. This plug-in together with a transformation and some third-party plug-ins are forming the architecture of the EAST-ADL modeling workbench. Since Papyrus is an open source tool, it is possible to attach any other transformation path here (e.g. EAST-ADL to Simulink®). With a generic transformation engine ATL (A Model Transformation Language) the architecture is transformed into an AUTOSAR model and finally deployed on to ARTOP (ATOSAR Tool Platform) which understands

⁶<http://www-list.cea.fr/>

2. Related Work

the model with respect to the AUTOSAR metamodel based on EMF. It is also important

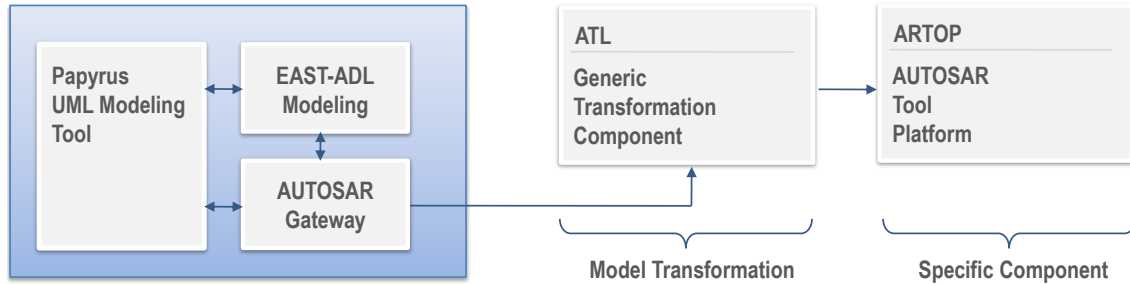


Figure 2.19.: EAST-ADL modeling workbench architecture, [SN10]

to mention that variability modeling aspects related to artifact level variability are a part of the core plug-in.

Regarding standardization, the UMLTM implementation in Papyrus fully conforms to OMG standards: XMI, UML and Diagram Interchange standard [SN10]. From the extensibility point of view, the plug-in provides the capability to connect a model with external tools (see Figure 2.19) in order to use the capabilities of various tools.

Analysis Platform This part of the tool environment in EAST-ADL is a combination of a set of plug-ins that can be used inside the modeling workbench for analysis purposes. In the following, the plug-ins are described shortly and sorted by severity based on project requirements.

Simulink Simulink plug-in provides transformation of Simulink models into EAST-ADL and vice versa. Generally, the conversion from Simulink to EAST-ADL model is performed by transforming a Matlab model into the intermediate, Ecore-based model representation which is in further step with usage of ATL transformation transformed into the EAST-ADL model. On a similar way the inverse transformation is realized.

Requirements Exchange Requirements exchange in Papyrus is performed by the *RIF Plug-in*. It allows the import of requirements into an EAST-ADL model and vice versa using the RIF (Requirements Exchange Format) standard for requirements interchange. The basic principles of requirements import in EAST-ADL can be described as follows: an existing EAST-ADL model file (XMI) with an RIF model (XML) are deserialized into models conforming to their metamodel specifications. These two models are now inputs for a M2M transformation. The result is a modified, serialized EAST-ADL model refined through new requirements. This works similar the other way round. The main problem here are specific requirements, which are not supported by the RIF standard (e.g. timing requirements). Therefore the EAST-ADL model has to be adjusted before performing the M2M transformation [MORST09].

AUTOSAR Gateway As mentioned before, the AUTOSAR gateway maps the design architecture described in EAST-ADL to the corresponding AUTOSAR component archi-

ture. The plug-in has been developed by CEA LIST in the scope of the EDONA project. It performs the mapping such that each EAST-ADL elementary function corresponds to an AUTOSAR software component (*Runnable*). Functions containing the functions are mapped to independent software components containing the *Runnables* [MORST09].

Safety Analysis The functionality of this plug-in is similar to that of the Simulink® plug-in. It transforms an EAST-ADL model into an HipHOPS Ecore model conforming to an HipHOPS metamodel. The second M2M transformation converts this HipHOPS Ecore model into the format which can be processed by HipHOPS, a tool performing failure analysis like FMEA [MORST09].

Timing Analysis This plug-in has also been developed by CEA LIST in the scope of the EDONA project. It enables the performance of schedulability analysis on existing EAST-ADL models i.e. by adding MARTE-based annotations to the design architecture model. The result is a specific format conforming to analysis tools which can then be used to present the analysis results [MORST09].

Summary Papyrus is an integrated tool environment supporting domain and application engineering in the scope of the EAST-ADL domain. It supports an EAST-ADL domain model fully, since it's development is synchronized with the ATESSST project. Therefore, the feature maturity criteria is satisfied here. Characteristics like extensibility and interoperability are not yet on the highest level, because some of the tools are still under development and available as prototypes only. On the other side, open source tools, which are easily extendable, are integrated in the release version of the environment.

A list of essential features provided by the tool as well as its drawbacks are shown in Table 2.3.

Selection criteria and (not-)supported features	
Advantages	Support for domain and application engineering
	Product derivation
	Extensibility (workbench only)
	Feature modeling and configuration propagation
	Variability modeling
	Feature metamodel maturity
Drawbacks	Technical environment (workbench only)
	Model comparison
	AOB (some of the tools are still under development)

Table 2.3.: Papyrus advantages and drawbacks

MetaEdit+ MetaEdit+ is a modeling and metamodeling tool developed in the scope of the MetaPHOR project at the University of Jyväskylä in early 1990s. Now it is a com-

2. Related Work

mercially distributed tool by MetaCase. It allows a definition of a new domain specific language with a very flexible tool chain. It further provides modeling means to use an instance of a created metamodel. A language is specified using the MetaEdit+® Workbench and a model is created in the MetaEdit® Modeler. The tool provides functionality to create generators for automatic code generation, to generate various reports and it allows a multi-user mode [Met] and [Has10].

As mentioned above, the MetaEdit+® Workbench is part of the MetaEdit+® tool environment and allows to create a new DSL by specifying its concepts, properties, rules, symbols and generators based on the GOPRR (graph, object, property, port, role, relationship) metamodeling language. The language concept defines the objects, i.e. elements to be used in a model, their properties (e.g. data types), relationships (e.g. inheritance) and constraints. Constraints are rules defining how elements can be connected [Met].

An essential feature of the MetaEdit+® Workbench is the possibility to define generators for code and documentation generation based on MERL, an internal scripting language (MetaEdit+® Reporting Language).

MetaEdit+® provides various ways to integrate an existing model into an external tool environment:

- SOAP interface: This interface can be used to create and modify models with a webservice client.
- XML import/export capability: Additionally to binary import/export capabilities, storing and opening of models in XML format is supported. This enhances exchangeability with external tools.
- Command line parameters allow to start a batch job, e.g. login, load a model, generate the code and export the model in XML format.

These characteristics make the tool highly interoperable and extensible [Met].

A distribution of MetaEdit+® comes with some exemplary metamodels. One of them is a EAST-ADL metamodel. The problem is that the EAST-ADL domain model is incomplete. Just the basic structural parts like FAA, FDA, HA and VFM are available. Another problem is the metamodel maturity. The EAST-ADL metamodel is very complex and changes, if necessary, are difficult to handle.

Summary MetaEdit+® is a leading tool for domain modeling and metamodeling with high interoperability, extensibility and flexibility characteristics. Unfortunately, there is lack of model comparison capabilities. Anyway, it could be used for domain engineering in EAST-ADL. The integrated EAST-ADL metamodel is not up-to-date, but a new one can be defined. It is also not easy to reflect all details of an EAST-ADL domain model, but at least it would be possible to create a metamodel containing the basic constructs required for the project or for special purposes of the first prototype. Additionally, the

metamodel can be extended at any time.

Another important characteristic is the use of a DSL for domain modeling instead of the FODA style used by most of the other PLE tools. Application engineering is realized by the use of modeling means (see [Has10]). Depending on the domain, this can be a complicated and time-consuming task. To overcome this problem, the tool could be complemented by e.g. `pure::variants` to satisfy the requirements of the whole system. This combination would be imaginable.

Table 2.4 shows the most important features provided by the tool as well as those that are desirable, but missing.

Selection criteria and (not-)supported features	
Advantages	Technical environment
	Product derivation
	Extensibility
	Flexibility
	Variability modeling
	Constraints checking and propagation
Drawbacks	Feature metamodel maturity (by specification changes, it is possible but difficult to update the whole metamodel)
	Feature modeling (no FODA style) and configuration propagation
	Model comparison

Table 2.4.: MetaEdit+® advantages and drawbacks

pure::variants Pure:variants is a commercial tool developed by pure-systems and shipped as an Eclipse plug-in. Generally, it supports domain and application engineering. Further does it separate both into a problem and a solution space. The product line architecture in this case would be a part of the solution space, i.e. it would be stored in form of a family model (representation of solution space). A domain may consist of several family models. Depending on the number of solutions (architecture, source code, tests, etc.). The family model acts as a kind of repository containing the artifacts that can be used to derive products with respect to a defined specification. The specification is described in the problem space. The generic specification (containing commonalities and variabilities) is represented in a feature model. Concrete variants are specified in a variant description model (VDM) which will be derived from the feature model. The VDM in combination with the family models result in a so called variant result model (VRM). In this model all the variability is resolved. It serves as the input for the product instantiation [ODF07] and [Lei09].

All models are stored in XML format and have the same structure. The basic construct is the *element*. An element can represent e.g. an elementary function in design architecture or the feature related to this function. To describe relations among the elements as well

2. Related Work

as restrictions defining the boundaries of some values, pure:variants provides restrictions, relations and attributes that can be added to an element. In the solution space (in the family models) there is a list of possible artifacts that an element can address, e.g. java file, another feature model, etc. These artifacts define the granularity of variants that can be provided by a family model [Pura].

There is also a connector for Simulink® available. One part of the solution is realized in Simulink®. There are blocks, for example *VAR_Constant*, defined to realize variant management. The real connector is realized as a pure:variants extension (Eclipse plugin). To use this plugin, a Matlab server instance has to be started to listen for requests from pure:variants. These requests are signals for a direct product derivation. In other words, these signals are sent to pure::variants blocks inside of a Simulink® model to change their parameters in order to derive a specific product. The result can be simulated at run-time. In this way the systematic reuse of Simulink® models and their constructs can be realized [Purb].

The tool has a very rich set of features that on one side allow to access and to control the models from external tools through high extension and data exchange capabilities and on the other side provides essential functionality related to product lines, i.e. model comparison, constraints checking and propagation, feature modeling, etc. [Purb] and [Lei09].

Summary Table 2.5 shows the features provided by the tool as well as drawbacks describing features that are essential for the project, but are not supported by pure:variants. To sum it up, pure::variants is a tool with a large arrangement of features related to product lines, but for EAST-ADL it is suggestive to use it only for the application engineering process. The support for domain engineering in EAST-ADL requires a model that corresponds to a very complicated metamodel.

Selection criteria and (not-)supported features	
Advantages	Feature modeling
	Technical environment
	Product derivation
	Extensibility
	Model comparison
Drawbacks	Constraints checking and propagation
	Domain engineering support for EAST-ADL Support for multi-level feature tree and configuration propagation

Table 2.5.: pure::variants advantages and drawbacks

openArchitectureWare openArchitectureWare (oAW) is an open source generator framework addressing model driven software development (MDD). It's currently developed in the scope of the EMF project (Eclipse Modeling Framework). The core of the framework

is the workflow engine which allows the definition of generator workflows based on transformations (e.g. M2M). It consists of three essential activities: definition of a metamodel, definition of the template and definition of the workflow. A metamodel corresponds to an *Ecore* metamodel based on EMOF (Essential Meta-Object Facility) and is strongly coupled with EMF. Since modeling with the standard *Ecore* editor is not comfortable, any other UML modeling tool, e.g. MagicDrawTM, can be used. The created UML metamodel can be transformed into the *Ecore* format using the UML2Ecore Utility⁷ for further processing (e.g. code generation). This is useful especially for more sophisticated metamodels. A template is a generator-specific feature which defines the rules for the code generation, whereas workflow is a kind of a batch job defining the sequence of executions in the code generation process (e.g. there could be more than one instance of a metamodel). Another way to define a metamodel is using the XText grammar language which additionally specifies the syntax for a DSL. Thus, the code can be generated from this textual representation with the template language XPand [Oaw].

oAW supports product lines analogous to the way MetaEdit+[®] does. The idea is to read the configuration containing a list of included and excluded features. The whole feature management, i.e. constraints between features and complex expressions must be managed by an external tool like for example pure::variants. To keep the consistency between the features after applying the new configuration, oAW provides a so called XVar tool which is aware of relations within a model. It is also possible to define the whole model as a single variant by using the oAW aspect weaver for models (XWave).

It is also important to mention that the *Ecore* format of an EAST-ADL domain model is available in the distribution of a Papyrus tool. This allows to import a metamodel directly into an oAW tool environment without any adaptations and without the definition of a new metamodel. In this way, it would be possible to define the grammar using the language capabilities to describe the model instantiation and to control it from an external tool. For instance, a tool for feature modeling management generates the specification corresponding to defined language which can be used by XPand to instantiate a model. This would be some kind of product derivation from the side of a feature modeling tool, which also must understand the syntax of that language. Thus, this solution would be imaginable.

Summary An oAW domain specific model conforms to the *Ecore* metamodel which is used in a broad community and can be transformed easily into other formats. For instance, the EMFText tool allows the textual representation of the *Ecore* models by using the transformation in both directions. Additionally, the model comparison is supported here.

The time effort to reflect an EAST-ADL domain model in oAW would be approximately the same as in MetaEdit+[®]. Contrary to MetaEdit+[®], oAW does not provide additional functionality like a SOAP interface for model manipulation from outside, report generators, etc. To solve these interoperability issues the tool could be extended. This seems not to be a serious problem since the tool is open source. Similar to MetaEdit+[®]

⁷A part of an oAW tool environment

2. Related Work

feature modeling in FODA style is not possible. Table 2.6 shows the most important characteristics regarding the selection criterias.

Selection criteria and (not-)supported features	
Advantages	Technical environment
	Extensibility
	Flexibility
	AOB (the tool is open source)
	Feature metamodel maturity (exactly like in Papyrus)
	Constraints checking and propagation
Drawbacks	Feature modeling (in FODA style) and configuration propagation

Table 2.6.: oAW advantages and drawbacks

Enterprise Architect Enterprise ArchitectTM from Sparx Systems is an commercial UML modeling tool addressing MDA, requirements and business process modeling as well as other more general modeling areas [Eas]. With a sophisticated graphical editor it allows to define a new DSL based on an UML profile or to use the existing one conforming to various versions of XMI format. Furthermore, features like generation of reports, comparing models and extensibility through add-ins are also provided by the tool. It is important to mention that the model comparison doesn't work for newer versions of XMI.

The multi-user mode is elaborated in a way that Enterprise ArchitectTM acts as a repository providing the models to its clients. Clients can either be instances of the *Eclipse* or *Visual Studio* modeling framework. For this purpose the plug-ins for *Eclipse* and *Visual Studio* must be installed. Additionally, the user working on Enterprise ArchitectTM can trace information about connected clients [Eas].

During the evaluation process each of the tools is inspected on ability to support an EAST-ADL domain model. Enterprise ArchitectTM provides functionality to import models described in various versions of XMI. But since there are various formats of the same version (e.g. Eclipse XMI 2.1 for UML 2.0, EA XMI 2.1, XMI 2.1 for UML2.0 MOF, etc.) there are incompatibility issues that result with the break-up of the import process. There is no way to log the error and to locate the problem in order to fix it. To overcome the problem, an alternative way would be the creation of a new metamodel with respect to an EAST-ADL domain model.

Summary Enterprise ArchitectTM provides a powerful graphical support for modeling, the ability for report generation, extension with DLL add-ins and interoperability through model export/import functionality. The problem is support for EAST-ADL domain model already available in XMI 2.1 version for UML2.0. The implementation of the metamodel

would be very time-consuming. Other tools support the EAST-ADL domain model directly (see 2.4.2.2).

MagicDraw UML MagicDraw is also a commercial UML modeling tool developed by No Magic for MDD, requirements modeling and other areas which are also covered by Enterprise Architect (EA). It supports M2M transformations, model validation, analysis functionality like model comparison, consistency validation and rich report generation [Mag].

The essential advantage to EA is the support of the *Ecore* format. It is not possible to import the *Ecore* model directly, but a tool can export any other supported model format to this one. A way to import a model directly is to perform the pre-conversion in EMOF, which unfortunately leads to losses in the model. Thus, there is also an integration capability which allows to work with models within the most popular MDA tools like oAW. This allows the conversions between UML and *Ecore* as well as transformations and generations provided by oAW [Mag]. An excerpt of an EAST-ADL domain model represented by MagicDraw is depicted on Figure 2.20. Concerning interoperability and

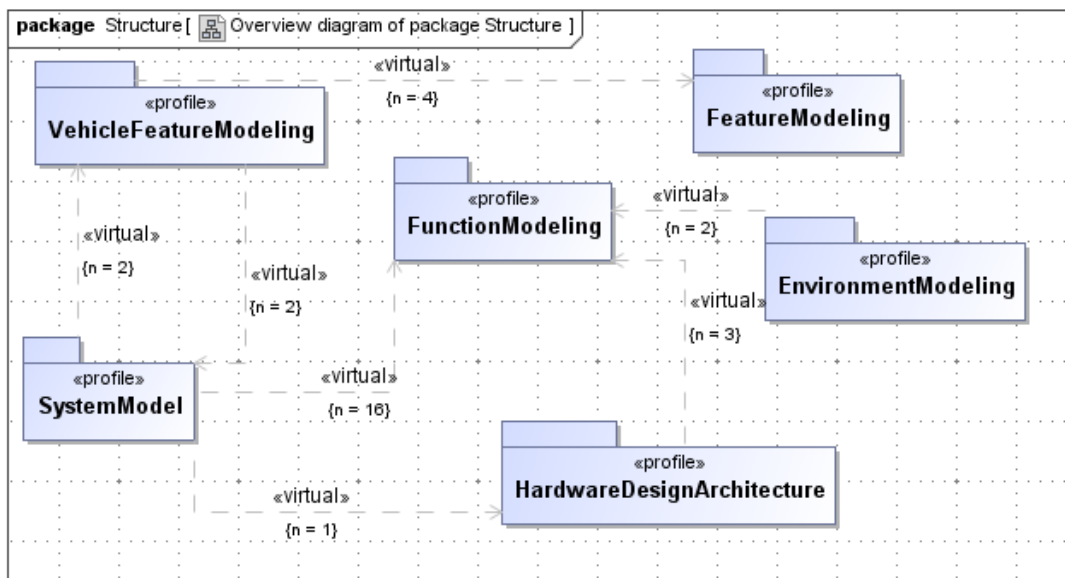


Figure 2.20.: EAST-ADL package as a part of UML profile extension imported in MagicDraw

extensibility, the tool is able to export models in various formats. More important, it allows the performance of model manipulations and transformations from external scripts written in script languages like Groovy, JRubby, Javascript, etc. Furthermore, distributed work like in EA is also supported here [Mag].

Summary Additionally to the features provided by EA, MagicDraw supports the exchange of *Ecore* (meta-) models and integration with oAW and other more generalized modeling tools.

2. Related Work

Common Variability Language CVL from SINTEF and University of Oslo is an variability modeling tool developed in the scope of the MoSiS project. Its purpose is the description of variability modeling means for frameworks, union of systems and domain specific languages in a common way with respect to the OMG standard. For now, it is available as a prototype only. The most important goals realized by CVL are:

- generic way to describe variability,
- intuitive way to configure products on a higher level of abstraction and
- automatic product derivation

CVL expresses the variability of any DSL conforming to the MOF or *Ecore* format as illustrated in Figure 2.21. On the right side of the figure is a model (e.g. *Ecore* EAST-ADL

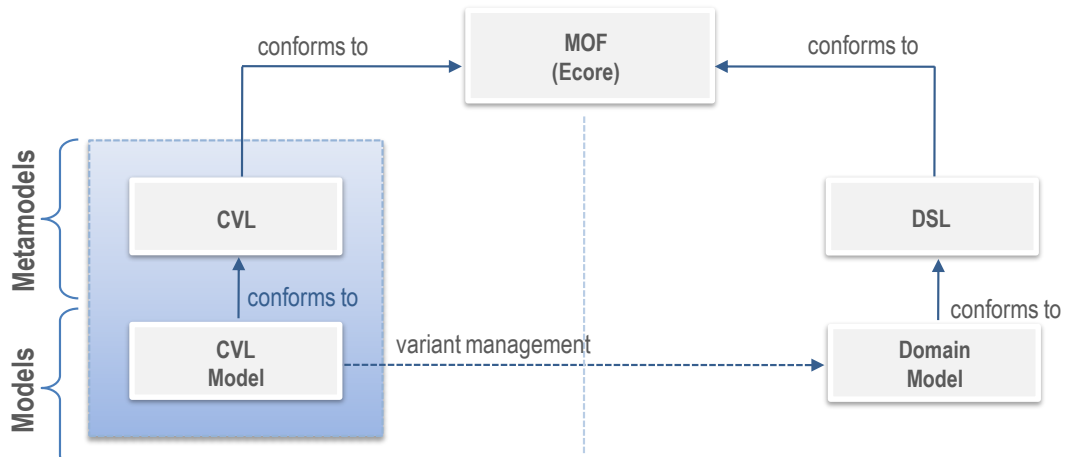


Figure 2.21.: Interface connecting CVL and DSL models, [Cvl]

domain model) which represents on one side a base model addressing the PL structure (FAA, FDA, HA and IA) and on other side a library model containing the software artifacts to be reused or replaced inside of a base model. Variability modeling means including feature models and artifact level variability are reflected in a CVL model conforming to an *Ecore* CVL metamodel. This approach allows to integrate the variability modeling in any model corresponding to *Ecore* or MOF without any dependency on an DSL tool [Cvl].

Generally, CVL distinguishes user-centric layer variability, which is similar to feature models on the VF level in EAST-ADL, and product-realization variability, which is analogous to artifact variability in EAST-ADL. Both variability means are modeled in separated editors provided by the plug-in. Modeling of user-centric layer variability is similar to the cardinality based notation of feature modeling as provided by EAST-ADL analysis tools. More interesting are technical aspects of product-realization variability. The main primitive used in product-realization level is the *fragment substitution*. To better understand the following explanation, it is advisable to see the Figure 2.22. The *fragment substitution* can be compared to variation points in the orthogonal variability model. The main idea is to replace a part of a model with another part. The degree of granularity may vary

from models to elementary parts of a model (e.g. variables, connectors, etc.). Base and library models are parts of a domain model described in a DSL. The first step is to define which parts of a base model are variation points or parts for replacement by adding the *replacement fragment*. Afterwards, the *replacement fragment* from a library model should

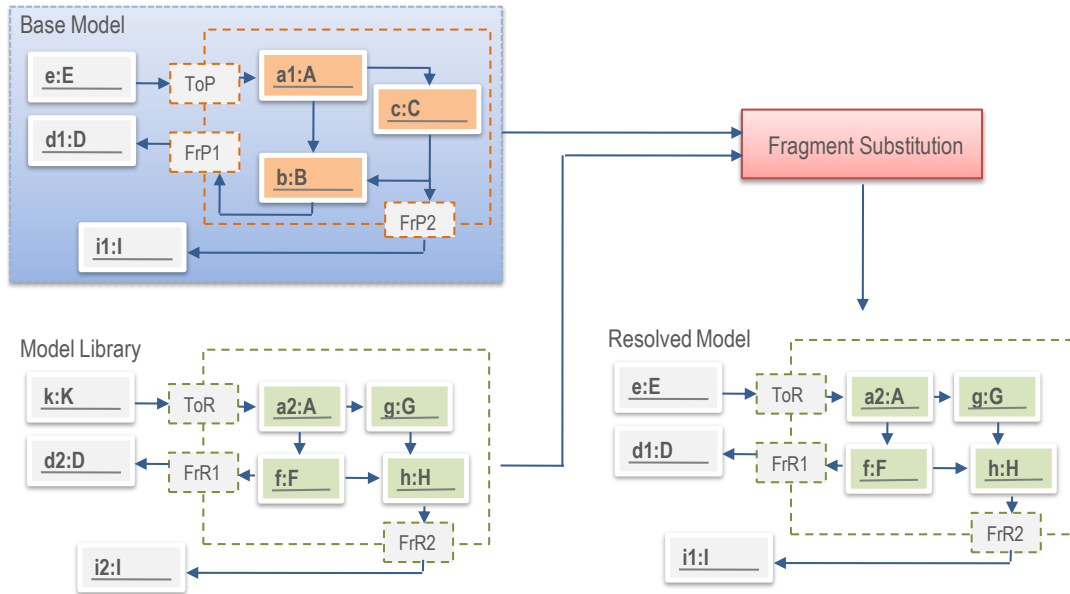


Figure 2.22.: Fragment substitution in CVL, [Cvl]

be created. This is a part of a model from the library which should replace the *placement fragment*. It is also important to be aware of the number, direction and types of connectors between the model and the *replacement fragment* (see Figure 2.22). At the end, the fine granular activities like variables replacement (for different variants) and fragment bindings need to be performed.

Product derivation is an automated process in CVL performed using the model transformation which additionally includes a resolution model which can be automatically generated by selecting the features from the user-centric layer. The result is a variability-free *Ecore* model.

Summary As mentioned above, an approach realized by CVL allows to integrate the variability modeling means in any DSL conforming to the *Ecore* format. Since an *Ecore* domain model of EAST-ADL is also available, it is imaginable to realize variant management in CVL. The remaining problem is that the whole variant management is already reflected by the EAST-ADL metamodel and any tool which supports this model is also able to model variability. The only missing feature is the product derivation, which is also missing in this approach.

Tools Excluded from Evaluation Process The tools included in the evaluation process and described above are just a subset of the tools addressing PL, MDA/MDD or DSL.

2. Related Work

This subset is pre-reduced based on the benefit analysis described in the two master thesis of Andrea Leitner [Lei09] and Andreas Haselsberger [Has10]. Tools from [Lei09] and [Has10] with a low weight distribution regarding domain and application engineering are excluded from the evaluation process (e.g. XFeature, DSL tools for VisualStudio, etc.).

For some tools an evaluation is not possible. They are also excluded. These tools are, for example, SystemWeaver and Vehicle System Architect. They are commercial and are not available in demo version at least.

2.4.2.3. Results of Tool Evaluation

Table 2.7 shows the evaluation results for the tools described in Section 2.4.2.2. The best matching tool is Papyrus followed by pure::variants, CVL and MetaEdit+®. This leads to the conclusion that criteria with actual weight distribution are more specialized for SPLE tools. Even the most weighted criteria are related to domain modeling, the features belonging to technical criteria have a worse ranking in metamodeling tools and degrade their results. Therefore, it is difficult to compare tools addressing different approaches. Concerning product line engineering, Papyrus seems to be the most adequate candidate since it provides the means for both, application and domain engineering management. Assuming that the criteria addressing application engineering are excluded from the evaluation process, or in other words, without the use of CVM Plug-in for Papyrus, the situation would look different. Pure::variants matches best for most of the evaluated criteria. The major drawback is that there is no direct support for the EAST-ADL metamodel. The differences of the results of the best suited tools are minimal. The differences from Papyrus to the results of pure::variants, CVM and MetaEdit+® are approximately 0,13%, 13,76% and 14,16% respectively. Especially a difference of 0,13% requires a more detailed evaluation of the two tools.

To still have the ability to compare the results with approaches described in [Lei09] and [Has10] based on criteria from Table 2.7, the second stage of the evaluation is derived from criteria described below. The idea is to perform the evaluation for criteria addressing the application and domain engineering separately in such a way that in each evaluation step only the tools corresponding to the proper engineering process are evaluated. The weights are calculated based on existing weight distribution from Table 2.7. First of all, the list of criteria is separated into three criteria groups: application engineering, domain engineering and criteria belonging to both engineering processes. Criteria addressing application engineering are product derivation and application engineering management, whereby attributes management, domain engineering management, model comparison, feature metamodel maturity, repository and impact analysis are related to domain engineering. For instance, to calculate the domain engineering weight value for Papyrus, all domain engineering criteria are multiplied with the corresponding weight factor and summed up. The results are shown in Table 2.8.

Now, the situation is much more better. Without adjusting the criteria and weight distribution, the evaluation could be performed giving satisfying results. The best matching tool for domain engineering is Papyrus followed by pure::variants and oAW. On the other side,

pure::variants leads ahead of Papyrus CVM and CVL. To sum it up, the evaluation process results with an integrated tool environment combined by Papyrus and pure::variants complementing the functionality for application and domain engineering.

2.5. Hypothesis

From now, topics handled in the related work should give a clear direction towards design and implementation of the proof-of-concept. The prototype described in the following sections should provide a variant management and reuse strategy for architectural core assets. As an approach for handling this, the SPLE has been chosen. This implies that commonalities and variabilities in the platform are created in the scope of the domain engineering process, which in addition could be (partly-) automated. The systematic reuse of these core assets is part of the application engineering process which is partly⁸ supported by an tool environment consisting of pure::variants and Papyrus. This combination is a result of the tool evaluation (see Section 2.4.2). Papyrus is, in addition, used to support the domain engineering process.

In Section 2.2 several ADLs are proposed and compared for their application in the automotive domain (see Table 2.2). The evaluation resulted with EAST-ADL which should provide a formal description of the software architecture. That means that all domain artefacts are in principle instances of EAST-ADL meta-classes, i.e. they form the platform. Since the UML profile of EAST-ADL already exists, a model of the software architecture is fully supported by Papyrus.

The last open discussion issue is the generation process of the software architecture. At this moment it is not defined how the control software is described, but somehow it should be reflected by its architecture. Therefore, it is to expect that the domain engineering process involves at least one model transformation. For this purpose, several approaches are discussed in Section 2.3.

To sum it up, the main challenge in this proof-of-concept is to extract the structural information from the control software and to build its architecture. To do this, a formal description of the control software has to be defined. One option for this description is AUTOSAR. Further, in correspondence to the captured variable parts of the control software, variability has to be generated. In addition, concerning the tool environment the possible alignment options between pure::variants and Papyrus have to be investigated. The aim is to support the whole development process in an integrated tool environment.

⁸Support for *Ecore*-based domain is investigated in later sections.

2. Related Work

	Tools	Papyrus	MetaEdit+	pure::variants	openArchitectureWare	Enterprise Architect	MagicDraw	CVL
Criteria	Weight	Ranking						
Attributes management	2	8	2	5	0	4	4	0
Feature / variability modeling	10	10	7	10	0	0	0	10
Domain engineering management	10	10	8	2	10	6	6	8
Application engineering management	7	10	6	10	4	3	3	10
Product derivation	8	9	5	10	4	1	1	10
Constraints checking and propagation	5	5	9	10	10	9	9	4
Model comparison	8	0	0	10	9	10	10	0
Feature metamodel maturity	7	10	4	2	10	8	10	10
Repository	3	6	6	6	6	4	4	4
Impact analysis	3	7	9	4	6	6	6	4
Reporting	4	4	9	8	1	10	10	1
Access mode	1	3	9	2	3	10	10	3
Technical environment	6	8	9	8	10	8	8	10
Usability	4	8	7	6	2	8	8	7
Automatic filters	1	0	6	5	0	5	5	0
Tool configuration	3	6	9	7	1	5	5	1
Extensibility	9	9	7	10	10	6	6	8
Flexibility	5	9	10	10	10	7	7	8
AOB	4	7	7	9	10	7	7	1
Summary								
PLE	600	471	327	442	370	290	324	432
Management	70	37	63	44	22	58	58	16
Technology	330	255	265	276	254	227	227	210
Overall summary	1000	763	655	762	646	575	589	658

Table 2.7.: Results of tool evaluation

	Tools	Papyrus (incl. CVM)	MetaEdit+	pure::variants	openArchitectureWare	Enterprise Architect	MagicDraw	CVL
Process	Max.	Ranking						
Domain engineering	850	621	573	612	586	546	560	508
Application engineering	700	559	525	620	386	359	359	496

Table 2.8.: Results of tool evaluation derived from Table 2.7

3. Design of the HybConS Architecture

This section deals with the technical realization of the HybConS architecture which is conceptually described in Section 1.1.1. At this point, it is necessary to map the concept to the target architecture. As mentioned before, software product lines are applied here to support the systematic reuse and variant management of core assets. Thus, the representation of the underlying domain reflected in SPLE framework (see Figure 2.2) is depicted on Figure 3.1. Here it is important to mention, that not all parts of domain and

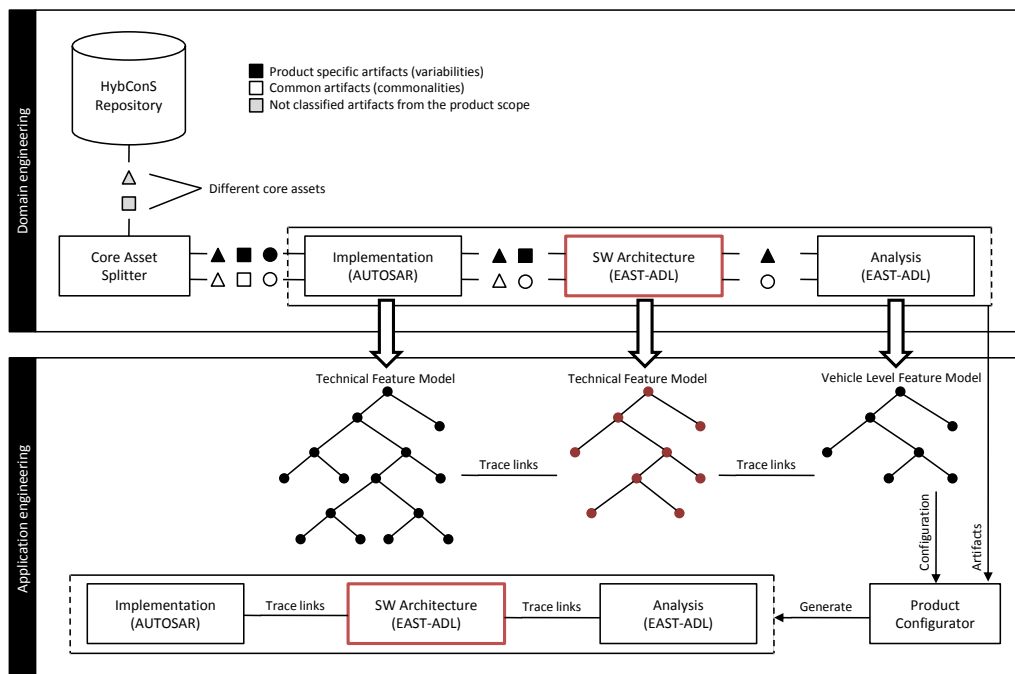


Figure 3.1.: HybConS processes reflected in SPLE framework

application scope are present (focus on architecture and implementation) in this picture. For now, they don't have a big influence on the architecture module. Another anomaly is the reversed order of core assets generation. From the fact, that the input for the platform (reusable core assets, see Section 2.1) generation process is a software or more precise the implementation assets, the generation is reversed, i.e. from implementation towards requirements.

The upper part of the framework is a domain engineering process which creates common and variable assets within separated sub-processes and provides everything required for their systematic reuse (see Section 2.1.1). For instance, one sub-process is responsible for maintenance of architectural assets only (upper red marked block in Figure 3.1).

3. Design of the HybConS Architecture

The source of these assets is the so called *HybConS Repository*. It is a collection of the control software components for hybrid vehicles described in Simulink. There are no commonalities and variabilites present, but just models with different realizations of the same “thing”. The “thing” can be seen as any core asset in the system which can vary.

For a specified range of products the separation of assets into common and variable parts is performed. This is one of the most important tasks for building the domain. This task is perfomed by the *Core Assets Splitter*. Ideally, this process is (partly-) automated, but it is still an open issue for future work. Furthermore, these variable and common parts are forwarded to the implementation module. The responsibility of this process is to map the incoming groups of core assets into the target model representation (formal description of implementation core assets) with the ability to describe variable core assets within the same model. Additionally, the functionality for variant handling and systematic reuse of these assets is provided. From now, all relevant steps in the platform generation process are completed.

In the next step of domain pipe (implementation, architecture and analysis), the structural information is forwarded and the architecture is created. Like in implementation phase, there is a target model representation conforming to the software architecture meta models. It should also be able to manage variants in the similar way as in the previous phase. This is important for handling traces between different variability models.

The developer performing the system configuration ideally works with the abstract representation of the system. This abstract representation identifies each variable asset within the architecture by propagating the configuration down to the FDA abstraction level. The next issue is to derive the abstract functional model from the architecture and to describe it with assets in the analysis block. Currently, this information is not explicit in the HybConS project, but it is probably the only way to get the feature model on the vehicle level. The importance of this feature model is reflected in the application engineering process. To configure the system, the product specification as well as core assets from the domain are required. The first corresponds to the vehicle feature model referring at least to the abstract functional system representation, i.e. analysis block. “At least” because features should describe what the system has, i.e. high level abstraction, and not design or implementation details. They are typically derived from requirements, and not from abstract functional model. But this feature model in combination with existing system requirements allows to adapt the vehicle feature model enough to be easily understandable for the system configuration. Here it is also important to mention that the existence of a technical feature model is not obligatory. There must be at least one technical feature model, referring to core assets in all abstraction layers which is configured by the vehicle level feature model as described in Section 2.2.2.5. If there are many feature models, e.g. one for each layer, the independence of models can be achieved and they can be developed separately from the whole system. This option is rational if the system is expecting a big amount of features (see Section 2.2.2.5).

The flow in the application engineering process is not reversed. Since domain artifacts

are available, this process can be performed as described in [ODF07].

The main goal of the domain sub-process for the architecture module is the generation of the product line architecture including the functionality for systematic reuse and variant management of architectural core assets. This involves also the interoperability with the implementation and analysis module. The following sections describe how this is realized.

3.1. Requirements

Regarding the process description given in the last section, it is now necessary to extract the most crucial technical requirements addressed to the architecture module. They can be defined by decomposing the project goals related to the HybConS architecture and analysing dependencies to other modules within the framework. The following content is related to Figure 3.1.

Development of domain sub-processes should have a predefined order, i.e. before starting the implementation part it should be clear what the *Core Assets Splitter* has to provide. It is difficult mainly in design phase of remaining modules in the pipe to decide how the *Core Assets Splitter* provides its goods. It gets more difficult if the development of these sub-processes is not synchronized. Namely, specifying the “how” in the “foreign” module leads to obligatory appliance of this decision in later development of the module. This may be a critical task, since the time investment in related topics is typically less than in research heap addressing the target module. These considerations are important, because the development of solution for systematic reuse and variant handling starts with the architecture. Until now it is known that the control software provided by the project partner is realized in Simulink and that the target architecture should be described in EAST-ADL.

Starting with the interoperability issues, the paths from and to the neighboring modules need to be considered. Namely, the responsibility of the implementation block from the architectural point of view is to provide the implementation assets. These assets need to be described somehow or at least the interface to the implementation block should be specified by some (standardized) template for the control software. Since it is not yet specified, one of the first steps in design is the definition of the software component description. The main characteristics of such a template are standardization and domain specificity. The later means, the templates describing software in automotive domain have a higher priority.

With the definition of the template for software component description the conflict with the implementation block is solved. Afterwards, the system should provide the mapping, i.e. detailed specification and realization which describes how to map the implementation core assets to the architecture in EAST-ADL. This can be decomposed into the analysis of both metamodels in order to extract candidate assets and finding the analogy between different model representations.

The next, and most scientific part of the project is to provide the variant management

3. Design of the HybConS Architecture

and systematic reuse of generated architectural core assets by following the methodology described in [ODF07]. The existing mechanism for variant handling in EAST-ADL should be exploited and compared for conformance to SPLE. The language therefore has to be evaluated for applicability in this project.

The last coarse grained requirement is provision of the interface to the analysis block in order to allow it to realize the abstract functional model from the architecture. This can be seen as an implicit task, but is also not unimportant, because this functional model is a key for the system configuration.

3.2. Scope

Before starting with how to realize requirements, it is advisable to define the product boundaries in means of product features. These boundaries define the product scope.

It is already known that the core assets of the implementation block are being mapped to the architecture described by EAST-ADL, but not which part of EAST-ADL. Namely, as elaborately described in Section 2.2.2.5, the engineering information is distributed over five abstraction layers. One of these layers is the design layer which is composed of the architectures describing the software (FDA), the middleware (MDA) and the hardware (HDA). For this project, only the architecture of the control software is generated. Hence, the elements of the FDA are used to build this architecture.

Another important aspect of the scope is to clarify what is supported by applied variant management with respect to granularity, binding times, feature models etc. Typically, by documenting variability some questions like what varies, why it varies, how it varies, for whom it is documented etc. are answered. This is especially essential if an already existing variant management mechanism is used (e.g. EAST-ADL variability). Therefore, this mechanism has to be inspected for these features. This leads to more detailed analysis and therefore it is documented in the next sections.

3.3. Domain Engineering

Further decomposition of requirements into use cases, user stories, etc. is not necessary, since the development is not team based for this project.

The purpose of the generator for the architectural block from Figure 3.1 is to build the reference architecture and the logic for the system configuration. In this section, the software design for this generator is explained in detail.

3.3.1. Variability Documentation

Documenting variability (see [PBvdL05]) involves a short analysis of assets of the EAST-ADL metamodel to answer the questions mentioned in the previous section. The first point is to define what can vary from the range of core assets in FDA. In EAST-ADL one

has to distinguish between asset types and prototypes (cf. component reuse in Section 2.2.2.3). Principally it should be possible to define each kind of prototype as variable element. Anyway, there is a predefined set of prototypes in FDA that can vary [Con10a]:

- FunctionPrototype
- FunctionPort
- FunctionConnector
- HardwareComponentPrototype
- HardwarePort
- ClampConnector

The combination of these prototypes allows to define variability on various levels of granularity. In the proof of concept the subsystems, software components and their specializations, i.e. ports and connectors, are included.

The input to the sub-process for the architecture module is an implementation model containing common and variable assets. In other words, the information to build the architecture is extracted from legacy code. How these variable assets are extracted from the implementation depends on the product scope and the mechanism hidden behind the *Core Assets Splitter*. So, for realization of this sub-process it is not important why these prototypes are varying. They are all internal variants. Their abstract representation is the result of the sub-process for creation of the analysis block. Typically, the whole process is reversed.

In the literature [ODF07] the term “artefact dependencies” is used to answer the question “how does it vary”. This is typically solved by creating traces between variants and domain elements. In EAST-ADL, or more precisely in the variant management part of the metamodel these traces are twofold. On the one hand domain assets are traced to variation points on the artifact level. This is known as artifact level variability. Different options of these variation points are realized by special metamodel elements named variable elements with direct traces to domain assets. On the other hand, domain assets are traced to the features and feature models. These feature models are also orthogonal to the assets model and have just references to them. Feature models are hierarchically ordered. The root feature model is called technical feature model. It is configured by the single vehicle feature model which belongs to the group of variability on the vehicle level. The combination of both levels of variability allows to define the product line and to configure the system in EAST-ADL.

Since the assets are captured from the implementation block, there is no ownership information. Therefore, all variants are set to be internal, i.e. *design artifact rationale* in EAST-ADL. This would mean, that the customer is not able to configure the red marked feature model from Figure 3.1. There is a need for further abstraction of the model. The benefit is that the abstracted feature model does not need any logic for configuration, because all variability logic already exists in the architecture. Therefore, just the traces to the architectural feature model are necessary.

3. Design of the HybConS Architecture

3.3.1.1. Variability in Design Assets

The architecture being generated here is in the literature [ODF07] known as reference architecture. Variation points within this architecture can adopt various forms of structural assets. As mentioned in Section 3.3.1 there are several granularity levels for structural variants. They are in [ODF07] distributed over three views of the architecture: development, process and code view. It is also important to see, which parts of these views are supported in the proof of concept, even if the topic is not explicitly referred to the automotive domain.

Development View This view on the reference architecture, as mentioned in Section 2.1.1.2, describes the variation points as sub-systems, components and configurations. For subsystems, this would mean to describe the system as relationships among EAST-ADL composite functions. Typically, most of the external features in the system are defined exactly here. A similar logic is applied to the components with the difference that elementary functions instead of composites are used. Furthermore, elementary functions are also not containers for connectors, i.e. relationships among components.

The situation with configurations is a little bit different. Namely, to provide variable configurations for components the injection of various realizations of component plug-ins is required. Since these plug-ins are in the target domain not required, there is no need to detect such constructs.

Process View Internal behavior of application software is also supported by the meta-model and various realisations of processes, i.e. different assignment options of threads to components are possible. But, since only structural information is captured from the implementation block, this feature is not necessarily part of the first prototype.

Code View This view deals with the distribution of the source code and executables over ECUs (cf. deployment diagram). Like in the previous view it is possible to capture the implementation of behavior. However, in the first prototype this feature is not intended to be present, because the mapping of software components to ECUs is out of focus for this work.

3.3.1.2. Binding Times

Binding times in EAST-ADL are introduced in Section 2.2.2.5. They cover the variants in all abstraction levels. There is a need to define a subset of binding times being used in the proof of concept, i.e. a binding time that address the architectural core assets only. Because there are only subsystems and components planned in this prototype, the system design time is required. On demand, other binding times can be supported in a similar way.

3.3.2. Product Management

In the product management the scope of the product line is defined and stored into the *HybConS Repository*. This is typically done manually by developers and project partners.

For other domain sub-processes there is no further work in this area, but it is important to know that two groups of assets (commonalities and variabilities) are a part of already predefined product scope.

3.3.3. Software Component Description

Before defining main design decisions for the domain engineering process, it is important to find an adequate description for implementation core assets. They are a basis for the software architecture in the HybConS. Moreover, an adequate mapping between this description and FDA has to be specified.

The first requirement related to domain engineering is the definition of the specification for description of software components (see Section 3.1). Thus, there are two options to solve this. On the one hand, the template can be defined in the scope of this project, on the other hand an existing template for the software component description can be used. Independent from the solution, at this point it is necessary to introduce domain assets from the FDA to find out what should be provided by this template.

The application software on design level in FDA is described by concrete functions called design function types. They capture the functionality provided by a particular software component. Because the application software is not just a collection of device independent components, there are some specializations of design function types: basic software function type, local device manager and hardware function type. The first specialization describes an abstraction of middleware, i.e. low level I/O API. The second, i.e. local device manager is usually used for translation of e.g. electrical signals into logical, understandable signals for application software. For example, if the temperature captured by the sensor is delivered to the microcontroller's I/O as a voltage, the local device manager is used to translate the voltage to the temperature and to deliver it to the application software component. For value calculation it uses the characteristics of the underlying hardware. The last specialization, i.e. hardware function type, can be seen as a transfer function of the real hardware. For instance, if the temperature sensor is giving the measured temperature as a voltage on its output, the hardware function type would describe the transfer function defining the transformation from the temperature to the voltage. All design function types a possible interaction is shown in Figure 3.2. The example shows the full path from the sensor to the actuator on design level interacting with the environment on the left side. Assuming that the sensor is capturing the temperature, the hardware function type *SensorF1* has to describe the transformation of the temperature into the voltage and to deliver the value to the basic software type *BSWSensor1*. It just forwards the value to the local device manager *LDMSensorF1*. From this point, the temperature is converted from the voltage to the real value and delivered to the application component *F1*. In a similar way, the information is propagated to the actuator.

The instances of design function types are communicating to each other by using three kinds of ports: function flow ports, client-server ports and function power ports. The function flow port is intended for data-oriented transmission. It contains a buffer that can be always overridden if new data arrives. The client-server port is used usually when

3. Design of the HybConS Architecture

sending control signals. It follows the client-server communication pattern, i.e. the client port can be connected to one server port only. The last candidate of ports is used to describe physical interactions between environment and the FDA. Therefore, it can be excluded from the consideration in this project.

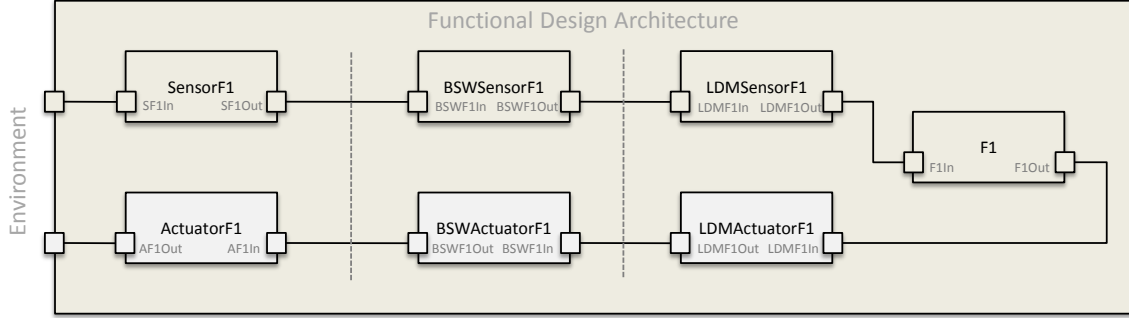


Figure 3.2.: EAST-ADL functions interaction on design level

Ports are transmitting data which is typed by either data types or interfaces. The first group is used by function flow and power ports, while the interfaces are a part of the client-server ports. They represent an abstract functionality behind the component by defining a set of provided operations. Similar to this pattern, a data transmitted between the flow and power ports is specified by a data type generalization named *EADDataType*. It supports the data representation as primitive types (*EAFloat*, *EAStrng*, etc.) and complex, i.e. composite data types. Additionally, a metamodel defines the constraints that must be respected in order to participate in the communication, especially if communicating ports are described by different interfaces or data types (see [Con10a]).

Furthermore, EAST-ADL provides different connector types for connecting ports. For FDA only one type is used and therefore there is no additional overhead for the mapping strategy to be implemented. Table 3.1 shows an excerpt of elements of the FDA captured in this short analysis.

It is important to mention that the properties of elements are also a part of the mapping process and they should not be ignored. Thus, detailed results of analysis can be found in Section A.

With this information it is now possible to create the template for describing the implementation assets and define the mapping to the FDA in EAST-ADL. It would be also imaginable to extract a part of the metamodel related to the FDA and take it as template metamodel, but the implementation block shown in Figure 3.1 would be EAST-ADL dependent. A better solution is to define an XML schema conforming to the elements from Table 3.1 and to the assets from the implementation block. In this way, necessary adaptations can be easily performed.

Different ways for describing components are summarized in Section 2.2.2. Before making a final decision existing techniques for component specification should be analysed.

Design function types	
Name	Description
Design function type	Composite or elementary function of the application software
Hardware function type	Transfer function of the hardware
Basic software function type	Middleware functionality
Local device manager	Functional description of signal calibration between BSW and the application software component
Ports	
Function flow port	Port for data-oriented flow
Function client server port	Port for client-server communication
Interfaces	
EA datatype	Data type used by flow ports to represent transmission data
Function client server interface	Set of operations provided by the client-server port
Connectors	
Function connector	Connector interconnecting two ports
Instances	
Design function prototype	Instance (prototype) of design function types

Table 3.1.: EAST-ADL2 elements used in FDA

Typically, the component description is loosely coupled with the architecture description. According to [SAB09] the architecture can be described either by using object-oriented notations (OBSA, Object-Based Software Architecture) or domain specific notations (CBSA, Component-Based Software Architecture). The component description template here is defined as an hybrid model (COSA, Component-Object-based Software Architecture) by combining both techniques: components, connectors and configurations are represented as classes (first and their instances can be used to build the model. Other constructs like interfaces, ports and properties are also covered by the metamodel. The realization of this metamodel is UML based. Thus, the consequences for applying this solution in HybConS would be almost the same as an option where an excerpt of the EAST-ADL metamodel addressing FDA is used as a template.

In [NXX10] and [LGLH08] a solution to describe software components by using an ontology based on a 3C¹ model is proposed. The goal is to improve reusability of software components. Typically, the usage of such semantic-based solutions in a domain with complex relationships among participants is advisable. Its main advantage is the navigation through the network in a domain with very simple queries. Instead, the functionality for each navigation step need to be explicitly implemented. In [NXX10] the schema was

¹Model for describing components for reuse with three attributes: context, concept and content [WR02]

3. Design of the HybConS Architecture

realized as a collection of generic components containing base information, interfaces, functions, environments and quality. The use of this template in HybConS would require adaptations in the domain model to support the automotive domain. Moreover, the communication interface between blocks from Figure 3.1 would not be “enough” application independent.

The standards from automotive domain, Fibex and AUTOSAR including their exchange data formats are introduced in Section 2.2.2.4 and 2.2.2.3 respectively. The AUTOSAR software components template would be more adequate since there are a lot of conformities to the EAST-ADL metamodel, e.g. same type-prototype patterns, similar communication pattern, similar types of component specializations etc. Moreover, EAST-ADL is seen as an AUTOSAR compliant architecture description language [CFJ⁺08]. Besides domain specifics, it is a widely used specification and closer to the standard as any other custom made solution. Since the template is described by an XML schema, there is no explicit dependence on a specific technology (e.g. in contrast to ontology-based solutions, [NXX10] and [LGLH08]). As a candidate for software component description for the implementation block, shown in Figure 3.1, the AUTOSAR software components template is chosen. Creating a new template similar to this one is not rational. One reason is the additional overhead to implement the XML schema. Another reason is the advantage of a standard solution compared to a custom-made solution. Table 3.2 shows the VFM elements from the XML schema corresponding to EAST-ADL FDA elements listed in Table 3.1.

3.3.4. Mapping Strategy

The mapping strategy concept for this project defines a detailed specification of how the elements from the AUTOSAR VFB are transformed into an EAST-ADL FDA. The FDA does not describe the software architecture from an implementation point of view, but rather the software from a design perspective. Since involved models are describing software on different levels of abstraction, there is a need for analysis of mapping strategies in order to generate a correct model (documentation) and to reduce losses in the transformation process. Figure 3.3 shows the black-box based representation of the software component and a possible alignment of various types of ports and corresponding port interfaces. It acts as a basis for further analysis and definition of the mapping strategy.

3.3.4.1. Existing Approaches

EAST-ADL documentation suggests two kinds of mappings: detailed mapping and black-box mapping. The first option should be used if behaviour of the target model is required. In this case, the design function corresponds to AUTOSAR *Runnables*. The other option is adequate if structural information is required only. Here, the design function is mapped to atomic or composite software component in AUTOSAR.

Until now there is only one practical realization of the mapping between EAST-ADL and AUTOSAR. In [OT10] and [AQST10] the project EDONA is introduced. The aim of this project is the integration of heterogeneous tools into one platform in order to support the cooperated development of automotive embedded systems. The essential part of EDONA

Software component types	
Name	Description
Application SW-C type	Elementary application software component
Composition SW-C type	Composite application software component
Complex device driver SW-C type	Special software component with direct access to hardware
ECU abstraction SW-C type	Software abstraction of I/O hardware
Parameter SW-C type	Software component acting as the container for parameters shared among other components
Sensor actuator SW-C type	Software representation of sensor/actuator
Service proxy SW-C type	Proxy software component allowing inter-ECU communication
Service SW-C type	Software component used for configuration of services on ECU (not a part of VFB)
Root SW-C type	Reference to root software component (instance)
NV block	Software component acting as the container for non volatile data
Ports	
Provider port	Port providing data
Requester port	Port requesting data
Interfaces	
Sender receiver interface	Interface representing the data-oriented transmission
Client server interface	Interface representing the control-oriented transmission
Connectors	
Assembly connector	Connector interconnecting two inner ports (inside of the same composite)
Delegation connector	Connector interconnecting inner port with the port outside of the composite
Instances	
Root SW composition prototype	Prototype typed by a root software component
SW component prototype	Prototype typed by any type of software components above

Table 3.2.: AUTOSAR VFB elements related to Table 3.1

for this project (see Edona V-Model [OT10]) is a transformation block named *ARGateway*. It transforms the software design described in EAST-ADL into an AUTOSAR software component description. Since the target model is located on the implementation level the use of a detailed mapping strategy is expected. The reason is, that not only structural information is important, but also behavioral information. However, two mapping strategies are proposed for design to implementation transformation:

3. Design of the HybConS Architecture

- M1 strategy: 1 EAST-ADL elementary design function is mapped to 1 AUTOSAR software component containing 1 runnable. Non-elementary EAST-ADL function is mapped to AUTOSAR composite software component.
- M2 strategy: 1 EAST-ADL non-elementary design function is mapped to 1 AUTOSAR software component. Elementary EAST-ADL functions are mapped to AUTOSAR runables.

3.3.4.2. Concept

Contrary to the mapping strategy proposed in the EDONA project, the implementation should be mapped to the software design in FDA. The mapping proposal in [CFJ⁺08] defines the FDA as the functionality of the application software architecture in AUTOSAR (its abstract functions). It also gives very important and helpful suggestions on detailed structure and behavior mapping. To begin with, one of the two mapping concepts proposed by EAST-ADL should be chosen. Since the structural information captured in the software architecture is of more importance than the behavior, the black-box solution is chosen. This level of detail excludes the mapping of runnable entities and reduces the overhead for realization of the transformation process.

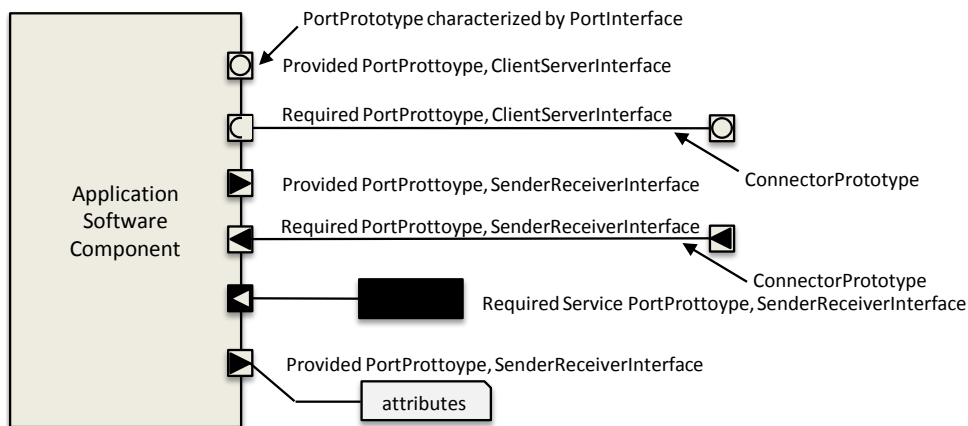


Figure 3.3.: Graphical notation of AUTOSAR software component, [AUT09b]

As previously mentioned, sources describing different mapping strategies do not provide a solution for a detailed mapping between EAST-ADL and AUTOSAR. Instead, they provide enough information about the similarity between models and possible mapping solutions that can be used to define the mapping. From now, there is a need for a detailed specification describing how elements and their properties are mapped. A candidate solution would be to find similarities between the two models describing the full information flow from a hardware sensor to the software representation of this specific sensor. In this flow, the most important (or all) groups of different software components may be included and can be therefore identified in FDA. Figure 3.4 shows the information flow on the example of capturing the velocity by application software components in AUTOSAR (see [AUT09b]). The physical value of the velocity is captured by the sensor and converted to

e.g. electric current. Typically, this signal is converted to a microcontroller input type e.g. voltage by *ECU Electronics*. From now, the signal can be captured by the standardized *HAL Driver* and is available in software. The next steps describe the reverse conversion. *ECU Abstraction* transforms the value from *HAL Driver* to “current” and delivers it to the software representation of the sensor. The sensor software component transforms “current” to the software representation of velocity [AUT09b].

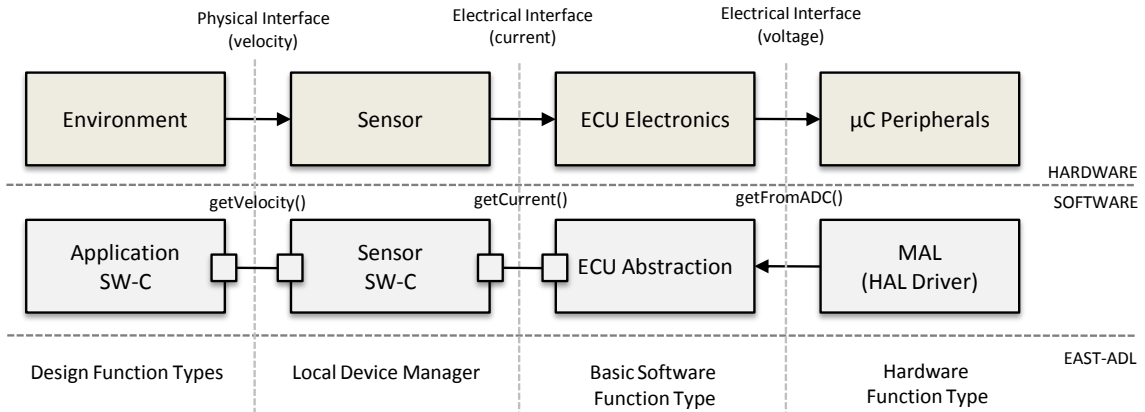


Figure 3.4.: Information flow in AUTOSAR [AUT09b]

Documentation of metamodel elements in combination with use cases like this allow to find analogies to EAST-ADL model elements. The corresponding classification of different design function types is depicted on the lower part of Figure 3.4. The detailed mapping specification based on material collected from [OT10], [AQST10], [CFJ⁺08] and [AUT09b] is described in Section A.

3.3.5. Variability Model

As previously mentioned the most scientific part of this thesis is the introduction of the variant management support for the architectural phase in domain engineering shown in Figure 3.1. Two possible realizations are proposed: (1) to implement the functionality corresponding to [ODF07] or (2) to implement the part of the EAST-ADL specification, which deals with variant management. The first solution suggests the implementation of such mechanisms outside of the EAST-ADL model whereas the second integrates this mechanism into the EAST-ADL model. Both solutions have advantages and drawbacks. The first solution is leading to model independence. For instance, if a new version of the EAST-ADL metamodel is integrated into the HybConS infrastructure, there would be no problems with this generic variant management mechanism. The problem with this approach is, that adequate variant handling addressing the automotive domain must be implemented from scratch. In other words, very intensive analysis and literature research in this area should be performed. It is not just a trivial task which binds traces between features and model assets. The solution (2) has already been implemented. Additionally, any tool implementing the EAST-ADL specification would be feasible to understand the product line mechanism and to configure the system without any built-in functional-

3. Design of the HybConS Architecture

ity. The only drawback is the model dependence. Each new version of the EAST-ADL metamodel would require some adaptations on the underlying mechanism. A conceptual description of this mechanism is given in Section 2.2.2.5.

3.3.5.1. Analysis of Variant Management in EAST-ADL

Following content is related to [Con10a] and [Con10b].

Before taking any decision, the variant management mechanism in EAST-ADL need to be analysed. Figure 3.5 shows an exemplary product line in the FDA and the corresponding orthogonal variability model including all model elements from the variability package (collection of metaclasses in the EAST-ADL metamodel for variant management). On the left lower side of the picture the model including two variation points (*Fb* and *Fb2*) is presented. The variability in this example is described in the development view (see Section 3.3.1.1).

In EAST-ADL there are two approaches how to describe variability: (1) variation points and (2) merging and optionality (see [Con10b]). The first approach describes different variants of the same asset inside of the variation point (similar to OVM in [PBvdL05]). Unfortunately, variation points have to be modeled inside the model. Another approach solves this by merging all variants of assets into the model. In other words, all variants are modeled without using variation points. There is a higher connectivity overhead, but this solution describes the variability independent of the model. Therefore the merging and optionality approach seems to be a more flexible solution for this project.

The right side of the figure shows a multi-level feature tree of the system. It is separated into a vehicle level feature model placed on top and public feature models addressing domain assets below. The red marked traces between feature models are just one possible configuration of decisions. They denote the effect of the feature selection in the source (referred) feature model. Thus, there is a freedom when defining a decision, because one feature on top can engage more than one decision when selected. For instance, selecting a single feature from top would affect the selection of multiple features on the artifact level.

On the upper left side of the figure, modeling elements of the variability extension in EAST-ADL are shown. They are also part of the generation process for variability extension. The following elements are used to build the variability logic in EAST-ADL.

Feature Model A feature model is a container for features. In EAST-ADL there are two types of such models:

- Vehicle level feature model: highest level of abstraction. In other words, this is the place for system configuration and the only place where customer visible features are present. The vehicle level feature model configures at least one technical feature model through the bridge called vehicle level configuration decision model. Features inside of this model can refer to the features of the technical feature model only.

- **Public feature model:** corresponds to the technical feature model, i.e. can be configured only by feature models from higher levels in the hierarchy (see Section 2.2.2.5). It describes the content inside a single composition. Features of this model can refer either to the features of referenced feature models or to model assets.

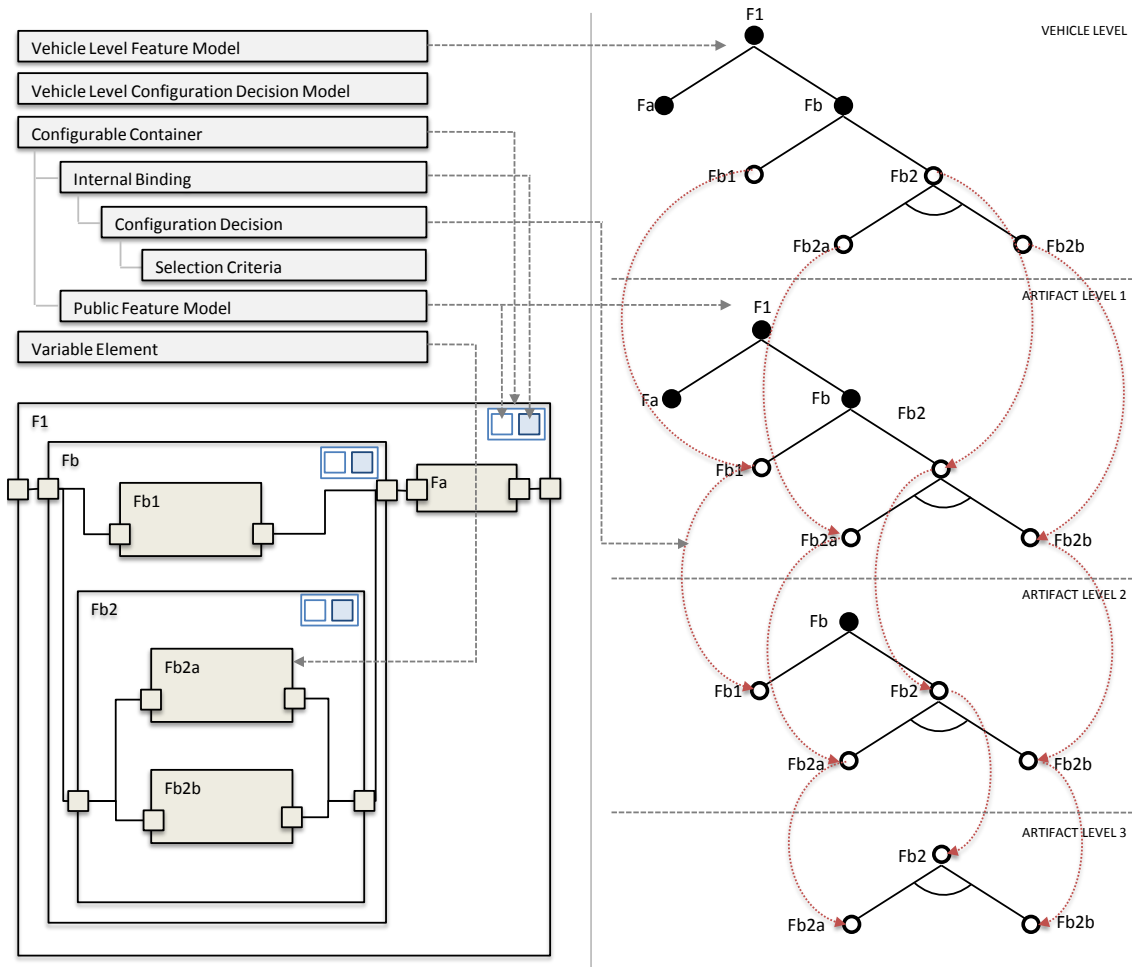


Figure 3.5.: Example of the product line in EAST-ADL: merging and optionality approach

Configurable Container The configurable container includes the configuration decisions for a particular composite. It holds the list of internal bindings, the public feature model providing the content in form of features and the reference to the composite being configured. In this way the independence between the variability extension and the model is achieved. Such kind of grouping of configuration rules allows to configure parts of the system without affecting others.

Internal Binding Internal binding describes the configuration inside a related composite by encapsulating its configuration decisions.

3. Design of the HybConS Architecture

Configuration Decision This is one of the most important elements inside the variability package. It describes how the parts of the composite are configured dependent on the selection of features in the source model. This single rule uses two attributes to make decision: criterion and effect. A criterion corresponds to an expression described in VSL (Variability Specification Language [Con10a]). It describes the configuration of the source model, e.g. (F1 & not F2) which means that an effect should be applied only if the feature F1 is selected and feature F2 is not. Effect describes the configuration of a composite in form of comma separated strings with the pattern <Name of FeatureModel>#<Name of Feature>. To keep uniqueness this rule provides references to the real model assets present in both attributes.

Vehicle Level Configuration Decision Model Similar to internal binding this configuration decision model encapsulates all configuration decisions related to the vehicle level feature model. It acts as the bridge between vehicle level and artifact level in EAST-ADL.

Selection Criterion The selection criterion is a part of the configuration decision and it holds the reference to the real model asset which is present in the mixed string expression in the criterion attribute.

Variable Element This is a single reference to the model element. It plays a prominent role especially in the merging and optionality approach. It makes it possible to distinguish between mandatory and optional model elements without affecting those elements having some additional attributes.

Container Configuration Container configuration is typically created as the input for the derivation process. It holds the configured content of a particular composite design function consisting of design assets and features describing this content. In other words, it describes the already configured composite.

Feature Model Configuration Similar to container configuration, the configuration of feature models is stored in the feature model configuration. The question is why to use separate configurations for the same composite design function. In Figure 3.5 configuration decisions (red marked traces) are used to connect features only. Therefore, they belong to the feature model configuration. The other group of decisions, i.e. configuration decisions related to variable elements, is not depicted in this Figure. They are responsible to bind the features that are related to design assets only. In Figure 3.5 this would be decisions for the features *Fb1* on level 2 and *Fb2a* and *Fb2b* on level 3.

Private Content One of the most important inputs for the derivation process is the knowledge about which design assets are a part of a new configured system. In EAST-ADL elements being excluded from the model are marked by the private content. This construct is similar to the variable element but it represents a different role. The derivation process filters the parts being referenced with the private content.

The model elements generated in the configuration process enable the creation of multiple system configurations. This is practical if more than one system specification is given at once.

3.3.5.2. Variant Management in HybConS

Applied Mechanism for Variant Management Since the variability mechanism of EAST-ADL is briefly introduced a decision has to be taken, if this mechanism should be used in this project. The other option is to implement a new variant management from scratch.

The main advantage of the EAST-ADL mechanism is “the whole behind it”. Namely, there are a couple of years of research in area of variant management in automotive domain. The project was started in 2001, but the variant management was introduced later. However, it is not imaginable that such a solution compared to the implementation from scratch provided in a couple of months, is not suitable for usage. But, there is also need to compare the candidates. Further advantages of the EAST-ADL mechanism could be summarized as follows:

- Independent development of sub-product lines (product sub-lines)
- Independency between variants in assets and feature models
- Model is fully independent from variability extension
- Tool independence for system configuration

The use of multi-level feature trees allows to develop variability logic for parts of the system independent from the root feature model or from the rest of the system. In this case, the feature models can be compared to interfaces defining the configurability of a particular composite. Therefore, integration of parts at a later point would not require any changes in the systems product line as long as the interfaces are not changed.

The next point relates to dependencies between variants in the product line. In EAST-ADL it is possible to build dependencies between features (include, exclude and inclusion criterion, see Section 2.2.2.5) and between variable model assets. This allows to handle internal variability which can be combined from dependent variable elements (e.g. optional software component requires the optional port). For instance, one selected feature from the vehicle feature model can affect at least one external variable element and null or more internal variants that are not visible for the customer or the system engineer. Without the ability to express dependency on artifact level, it would not be possible to handle internal variants. Thus, in this case they had to be defined as external features. For instance, assume that the software component *Fb2b* contains three variable parameters: p1, p2 and p3. The first two parameters (p1 and p2) are part of the component if *Fb2a* is also present in the system, otherwise, parameter p3 is included. This can be realized by extending the feature model on level 3 for the three mentioned parameters and by building dependencies to the software component *Fb2a* with inclusion criterion. The vehicle feature model is not aware of these details. This is an important characteristic of this multi-level feature tree approach. Now, the same with another kind of dependency, i.e. on artifact level would be

3. Design of the HybConS Architecture

to define more advanced VSL expressions inside the configuration decision connecting the feature *Fb2b* on level 2 and level 3 to select attributes p1, p2 and p3 in correspondence to the feature *Fb2a*. The criterion should be defined as follows: $(Fb2b \ \& \ \text{not} \ Fb2a)$ with the effect: $(PFM_Fb2\#Fb2_b, \ PFM_Fb2\#Fb2_b.p3)$. Similar to this, the second configuration decision is created for criteria: $(Fb2b \ \& \ Fb2a)$. Attribute p3 is used in the effect, because the pattern used in HybConS allows to express assets in this mixed string, even it is not explicitly defined as a feature (see Figure 3.6).

Typically, when realizing a domain model in EAST-ADL the whole content of the variability logic (feature models, configuration decisions, etc.) is present in the model. This implies that each tool implementing the EAST-ADL specification is able to understand this variability logic, to read it and to configure the system without requiring any built-in functionality. This would mean, that the complete product line can be delivered to the customer which can accordingly configure the system or parts of it by himself. Another use case is to deliver a part of the system to a supplier. The customer can extend or define the variability logic which can be integrated into the system at a later point.

Another proposal for variant management in HybConS, i.e. implementing it outside of the model, has as one advantage the independence from the model. It would be a generic solution that would work for any EAST-ADL metamodel version. But as mentioned before, there is a big effort for designing this solution for the automotive domain and therefore the first options is chosen for this project.

Configuration Rules The distribution of configuration decisions from Figure 3.5 is not the only possibility. There is a high freedom to trace decisions as long as their effect results in a valid model. But for the generation process, a common way to generate such decisions should be defined. Variable implementation assets are identified by the AUTOSAR schema elements related to variant handling described in Section 2.2.2.3. For each variant, one feature inside the composite's feature model is generated. This feature model is furthermore related to the feature model in the upper level of the hierarchy (see Figure 3.5). That means that the feature model generation is based on the hierarchy of sub-systems. To generate the variability logic for such a concept it is enough to have a decision distribution like the one shown in Figure 3.5. Each feature from the reference model (level L) configures only one feature or model asset in the referred model (level L-1). This leads to the definition of a single rule as depicted in Figure 3.6. It can be seen as a transfer function that for a given criterion x , including a single feature, results in an effect y describing the selected feature or model asset. All configuration decisions in HybConS are computed in this way. This kind of decision definition allows to model variability like the one depicted in Figure 3.5. However, decision criteria can be modified to provide more complex VSL expressions defining multiple dependencies. It is difficult to provide a generic solution for such complex expressions and therefore further dependencies should be manually modeled if required.

Deviation Set Another point where the freedom of choice for realization is present is the derivation of the feature models. The multi-level feature model shown on the right side of Figure 3.5 is just one possible solution for deriving referred feature models. For

instance, the last feature model on level L3 can be further decomposed to deal with more fine grained variability. Such 1 : 1 mapping between hierarchical feature models is known as *plain propagation pattern*, [RKW09].

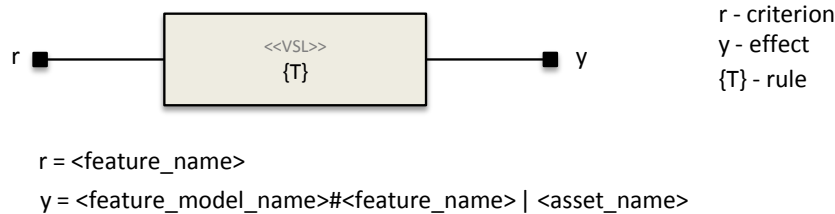


Figure 3.6.: Transfer function of a single rule in HybConS variant management mechanism

The freedom of decomposition is bound by a so called deviation set containing nine derivation rules. This set of rules ensures that a later integration of independently developed sub-feature models into the whole system can be done without problems. The rules are described in Table 3.3 with the configuration values used in this project. With this configuration, the reference feature model is just cloned and configuration decisions are created for each reference and referred feature. This can be later manually changed on demand.

3.4. Application Engineering

Application engineering deals with reuse of design assets in order to build the system conforming to a given specification. Again, there are two possible realizations for this process: (1) by using generated configuration decisions (EAST-ADL approach) and (2) by exposing traces to the external tool for system configuration. In Section 2.4.2 (Tool Selection and Evaluation) `pure::variants` has been chosen for system configuration, i.e. for application engineering. Therefore there is an option to provide an API which can be used by `pure::variants` to import the technical feature model and to build the family model. In this case, there is no need to evaluate each configuration decision inside a model and also there is no need to use the configuration mechanism of EAST-ADL.

Taking the first option in account, i.e. EAST-ADL variant management, each configuration decision needs to be evaluated and corresponding to the evaluation results, domain assets conforming to the specification can be extracted from the model in the next process (in system derivation). Thus, this option does not require additional overhead for the realization of the API, except of the provision of the vehicle feature model. Here, the mission of `pure::variants` is to provide the feature model which can be used by the system developer (or customer) to generate the variant description model. This model is then the input for the configuration mechanism of EAST-ADL.

The ideal solution would be to keep both options available. The EAST-ADL configuration mechanism can be used as “portable” functionality, since it is always present in the model. As previously mentioned, it would be possible to use any tool implementing the

3. Design of the HybConS Architecture

EAST-ADL specification in order to read and configure the model. In this way the system can be delivered to the customer without a tool environment. As long as this is not the case, the common configuration functionality of the HybConS environment is used. To

Name	Value	Description
allowChangeAttribute	yes	Definition of attributes of the feature in referred feature model during derivation (yes = attributes are derived from reference model, no = attributes are not derived, append = attributes are changed by appending value from the reference feature)
allowChangeCardinality	yes	-
allowChangeDescription	yes	-
allowChangeName	yes	-
allowMove	no	Movement of the feature in diagram. This is not required for proposed solution
allowReduction	no	-
allowRefinement	no	Refinement can be performed later manually
allowRegrouping	no	-
allowRemoval	no	-

Table 3.3.: Deviation set in HybConS, [Con10a]

support both solutions, the system configuration process for the architecture block from Figure 3.1 is realized by following the EAST-ADL specification.

3.5. Final Architecture

Figure 3.7 shows the logical view of design captured from analysis and decisions described in the previous sections. To demonstrate the most important parts of the information flow the implementation block (see Figure 3.1) is depicted on the left side. It holds the whole implementation assets and provides them through the AUTOSAR software component description template. There is a trace between the AUTOSAR schema file and the EAST-ADL metamodel named *partly conforms to*, which is present due to analogies between models described in Section 3.3.4.

On the right side, an integrated tool environment handling the architecture block is shown. Actually, the implementation block is part of this environment too, but it is out of scope for this thesis and therefore depicted outside. In short, the right side is responsible to produce various variable-free architectures of systems (products) for a given implementation model and product specifications.

The mapping process is a model transformation, which maps implementation assets described in AUTOSAR into an EAST-ADL FDA. Typically, when generating products by using approaches like software product lines, it is expected that the generated product does not realize all requirements by 100%, except that each requirement is identifiable

by existing features (see *deltas* in Section 2.1.1.3). But this is the ideal case. After the generation process it is required to finish the implementation by using traditional software development methods to satisfy remaining requirements. This is why the curve of the

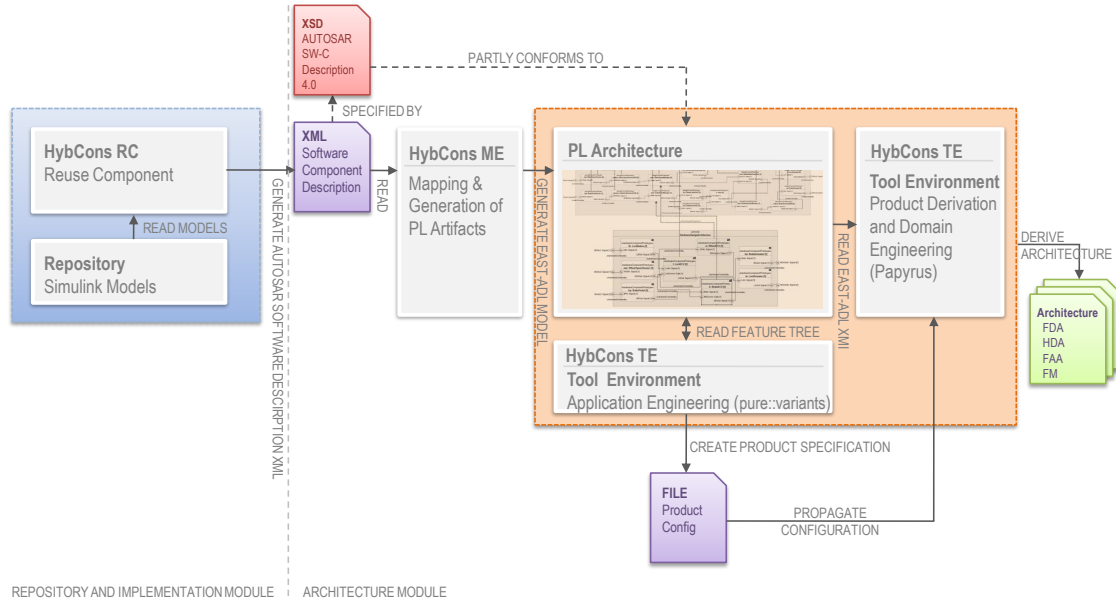


Figure 3.7.: Design flow of domain sub-process for HybConS architecture

graph for software product lines in Figure 2.1 is not horizontal. The implementation of *deltas* at a later point in time is supported by Papyrus. Hence, it is possible to refine and extend the generated variability logic for functionality that was unable to be generated as well as to adapt the configured system to satisfy all requirements. Papyrus is also intended to be used to make trace links between implementation assets, their corresponding architectural assets and the like. It is able to read the generated EAST-ADL model directly.

After model transformation, *pure::variants* reads the vehicle level feature model from EAST-ADL, which is placed inside a generated variability extension. This feature model is then a part of the problem space in *pure::variants* and it is configured in the next step. The configuration result is a variant description model which corresponds to the system specification. Assuming that the EAST-ADL configuration process is used for the system configuration, the specification is forwarded to this process without being aware of the family model. For another solution, i.e. when the family model of *pure::variants* is used instead of the EAST-ADL configuration decisions, the trace links to model assets need to be extracted from FDA in order to build the family model. To allow this, an API inside of the architecture block need to be provided.

4. Implementation of the Prototype

In this section the implementation of domain and application engineering is shown in form of three processes: mapping, variability generation and system configuration. In order to show the interaction of all processes, a simplified version of the design from Figure 3.7 including the main steps of the processes is depicted in Figure 4.1. The *Software Component*

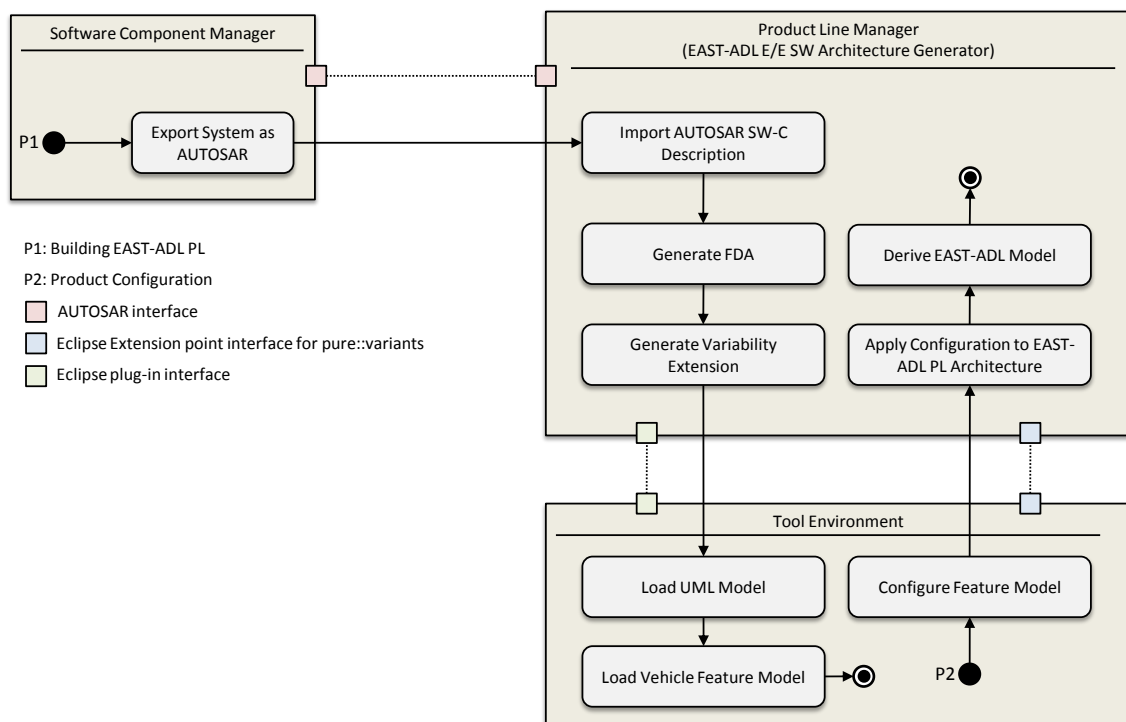


Figure 4.1.: Simplified processes for architecture generator in HybConS

Manager corresponds to the implementation block shown in Figure 3.1. It communicates with the *Product Line Manager* (architecture block) through AUTOSAR SW-C template. Both parts are actually inside a different tool environment. The interface *Eclipse plug-in interface* is used to import the model into Papyrus. The *Eclipse extension point interface for pure::variants* allows to communicate with pure::variants in order to read the vehicle feature model as well as to provide a variant description model (see Section 2.4.2.2), which is a basis for the system configuration process (see Section 4.2.1).

The mapping process (model transformation, see Section 4.1.1) P1 is engaged by importing the AUTOSAR file. The first step is the generation of the FDA corresponding to the mapping strategy described in Section 3.3.4. After performing the model transformation

4. Implementation of the Prototype

it is up to the developer to decide if the variability extension should be generated. If he confirms, the variation points captured from AUTOSAR are evaluated to identify variable assets. Accordingly, the variability logic is generated. The last activity in the process is to provide the vehicle level feature model to pure::variants. This can be only done if the variability extension is generated. In this way, the product line architecture can be configured even if the feature model is technical, i.e. it consists of internal features only. In EAST-ADL they are marked with the property *isDesignVariabilityRationale* to denote that they are not intended to be a part of the vehicle level (see Section 3 for discussion about this issue).

The configuraton process is engaged when the variant description model is created (if pure::variants is used as configuration tool). According to this product specification, all generated decisions are evaluated and only those conforming to the specification are candidates for the next stage, i.e. for the derivation process.

The derivation process here is a part of the configuration process because it is automatically executed after all configuration decisions conforming the the specification are identified. The aim of the derivation process is to read configurations generated in the configuration process and to generate the variability-free system. Thus, there is no need to separate these processes as long as there is no need to make different configurations before deriving the architecture. In this process, the relations to the real model assets are extracted from collected configuration decisions to identify excluded model assets. The derived architecture is built by removing these assets from the product line architecture.

4.1. Domain Engineering

4.1.1. Part I: Mapping Process

In order to capture implementation details, this process is described from two different perspectives. The development view typically shows the design of the system in the class diagram, whereas in the process view the most important object interactions in form of sequence diagrams are described. But here, only the major steps of the process as shown in Figure 4.1 are described in more detail.

4.1.1.1. Development View

Figure 4.2 shows the simplified class diagram of the architecture block from Figure 3.1. Each class, except of the *AutosarSAXParser* and the *AutosarModelBuilder* is instanced within *application scope* and therefore is accessible by the client all the time the application is running. This allows to work with the EAST-ADL model after its generation. The class instantiation is controlled by the *Spring Application Context* in bottom-up order. This application context allows to change implementation of classes (in the next: beans) without big effort (see Section A). The beans are systematically designed for replacement.

On the lowest level of the tree (bottom part in Figure 4.2), there are three registry beans. They are responsible for initialization of the system registry with the UML, the SysML

and the EAST-ADL metamodel. The system registry is an internal data structure for holding a so called *ResourceSet* from EMF (Eclipse Modeling Framework) which is used to manage a collection of resources, i.e. constructs of UML, SysML and EAST-ADL in this case. The metamodel path may be specified within the configuration and thus metamodels can be switched without affecting the implementation. These registries are a part of the container on the next level, i.e. *ResourceManager*. Its purpose is to initialize registries and to provide primitives for model loading and export.

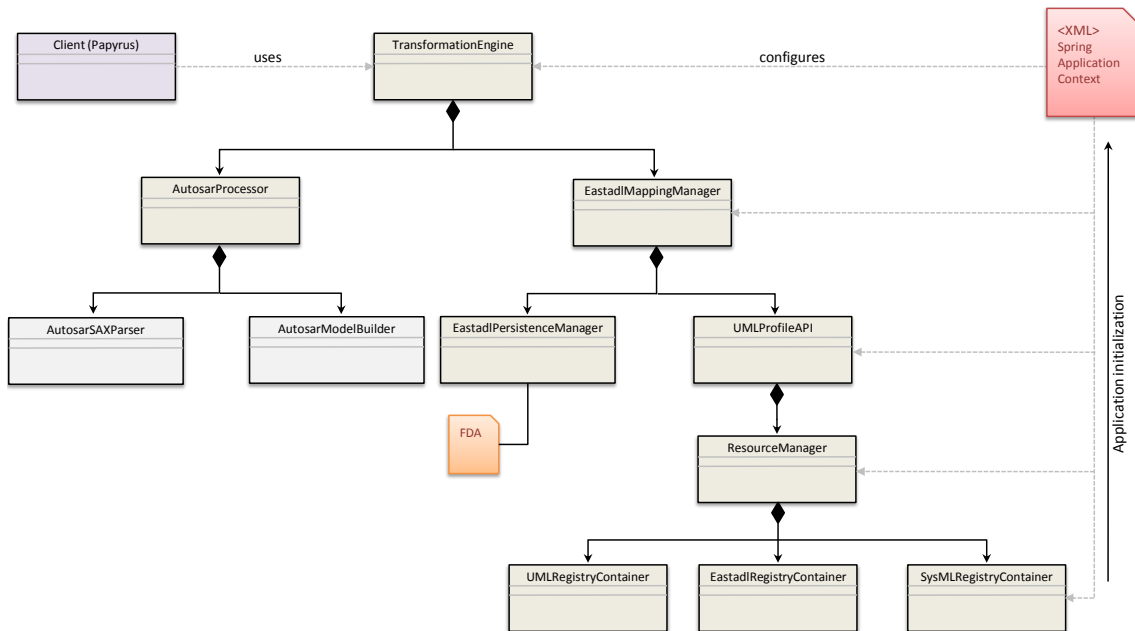


Figure 4.2.: HybConS architecture generator: development view

Now from the top. The *TransformationEngine* is an interface provided to the client (e.g. Papyrus) in order to execute the transformation or to provide access to the existing EAST-ADL model. It has access to all other beans in the application context and thus it can provide their functionality to the client. For instance, if it is required to export the model manually, only this interface needs to be extended. For model transformation two beans are used: *AutosarProcessor* and *EastadlMappingManager*.

The *AutosarProcessor* provides the generic representation of the AUTOSAR model captured by the *AutosarSAXParser*. This generic representation allows to attach different versions of the AUTOSAR schema into the transformation process. The bean *AutosarModelBuilder* is responsible for the creation of such a schema independent model.

The *EastadlMappingManager* is a collection of methods, implementing the concrete transformation according to the mapping strategy described in Section 3.3.4. To make the *TransformationEngine* independent from the EAST-ADL version, the mapping API works with UUIDs (Universally Unique Identifier) generated by the *EastadlPersistenceManager*. If the client really requires an EAST-ADL element, it can get it from the internal cache of

4. Implementation of the Prototype

the *EastadlPersistenceManager* by providing the generated UUID. Here is the only place to modify or extend the mapping if required.

The *EastadlPersistenceManager* is one of the most important beans within the application context. It provides all methods required to build model elements in the FDA as well as model elements from the variability package of the metamodel. For the creation of these elements the *UMLProfileAPI* is used. The process of creation is as follows: the stereotype from the metamodel is requested from the *ResourceManager* by bypassing the *UMLProfileAPI*. Then the primitive provided by the *UMLProfileAPI* is used to apply this stereotype on the created class or package. In this way, there is no direct dependency on the EAST-ADL version, except of hardcoded names of stereotypes within a metamodel. The effort to attach a new version of EAST-ADL is therefore relatively low.

The *UMLProfileAPI* consists of several primitives for the creation of classes, properties, packages and other metaclasses the EAST-ADL extends. Additionally, this path provides access to the EAST-ADL model in the file system, whereas the other path, i.e. the left side of the *EastadlMappingManager* holds the model in memory. Currently, this API uses version 2.1 of Ecore based UML.

4.1.1.2. Process View

The transformation path from the AUTOSAR software component description to the EAST-ADL FDA is depicted in Figure 4.3.

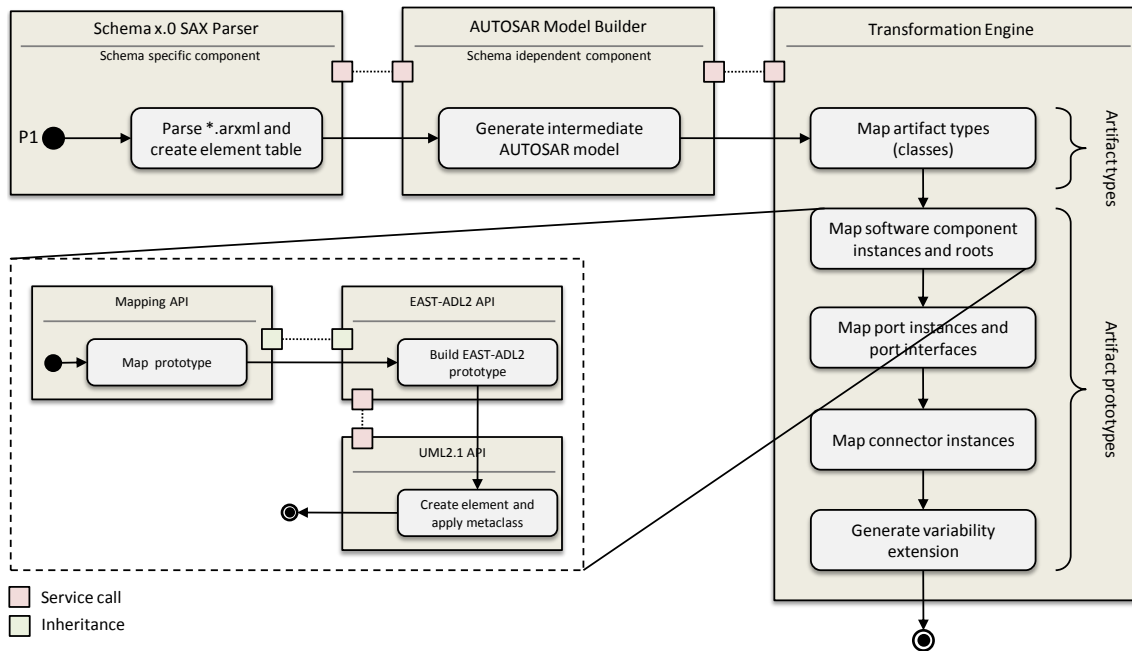


Figure 4.3.: Mapping process

For the specified AUTOSAR schema version, the corresponding SAX parser implementation is internally selected and the model is parsed. The result of parsing is an element table, which contains captured content. The purpose of this table is to estimate the *AUTOSAR path* for each element (see Section 4.1.1.2). Without this path, there is no chance to find references between elements. Like in EAST-ADL, the AUTOAR schema elements are of depth two. It is also important to mention, that the versions 2, 3 and 4 of AUTOSAR are supported. The versions 2 and 3 are intended just for test purposes and not for use, because not all necessary parts are mapped and there is no calculation for relative paths (see Section 4.1.1.2).

The next activity takes the element table and builds a version independent AUTOSAR model based on the concept of the software component representation depicted in Figure 3.3. Assuming differences between the AUTOSAR schema version for software component description, there is no change in the concept and very probably there will be no change in the concept for upcoming versions. The changes are basically related to the representation of metaclasses (e.g. different notation of delegation and assembly connector). Therefore the functionality behind the source model is more flexible when following this way, even there are inevitable but not significant losses in performance.

In the next step, the *Transformation Engine* uses the mapping API to build the EAST-ADL model according to the mapping strategy defined in Section 3.3.4. First it maps types (see types and prototypes in Section 2.2.2.3). A type may be e.g. a composite software component. The instance of this type is the prototype. Therefore, mapping prototypes require the existence of their types in EAST-ADL (cf. an object is not older than its class). In the next step prototypes are mapped. Since AUTOSAR provides constructs to express the reference to the root software components explicitly, these references are mapped as root prototypes within the FDA, i.e. they are first-level functions in the FDA.

The next candidates for mapping are ports and port interfaces, because now all software components already exist in EAST-ADL. The same applies to connectors. Since they are referred to ports, they need to be mapped at the end.

Finally, the variability extension is mapped. This process is elaborately described in Section 4.1.2.

Resolving Paths The reason for building an intermediate AUTOSAR model is twofold. On the one hand it is better to provide more flexibility when attaching new schema versions in the system, on the other hand, the paths which are used to identify AUTOSAR elements are not explicitly defined for each element in the model. Instead, they need to be calculated. Therefore, referencing elements in the first iteration would not work, because some references are processed later as they are required. Thus, a second iteration is necessary. Therefore, it is rational to use this chance for building the schema independent model in the second iteration.

Elements in AUTOSAR are encapsulated inside so called *AR-PACKAGE* constructs (see line 2 in Listing 4.1). On line 9 the package *system* is defined. Its purpose in this example

4. Implementation of the Prototype

is to hold the reference to the root software component. Listing 4.1 shows an excerpt of an AUTOSAR model.

```
1 <AR:AUTOSAR S="String" T="0000-00-00" xsi:schemaLocation="http://autosar.org/schema/
  r4.0 AUTOSAR_4-0-1.xsd" xmlns:AR="http://autosar.org/schema/r4.0" xmlns:xsi="
  http://www.w3.org/2001/XMLSchema-instance">
2 <AR:AR-PACKAGES>
3 <!-- top level packages-->
4 <AR:AR-PACKAGE S="String" T="0000-00-00" UUID="String">
5 <AR:SHORT-NAME S="String" T="0000-00-00">autosar</AR:SHORT-NAME>
6 ...
7 <AR:AR-PACKAGES>
8 <!-- root of sw-c : the complete system-->
9 <AR:AR-PACKAGE>
10 <AR:SHORT-NAME>system</AR:SHORT-NAME>
11 <AR:ELEMENTS>
12 <AR:SYSTEM>
13 <AR:SHORT-NAME>HybConS_TE_UC1</AR:SHORT-NAME>
14 <AR:ROOT-SOFTWARE-COMPOSITIONS>
15 <AR:ROOT-SW-COMPOSITION-PROTOTYPE>
16
17 <AR:SHORT-NAME>CM2ARSystem_autosar1</AR:SHORT-NAME>
18 <AR:SOFTWARE-COMPOSITION-TREF DEST="COMPOSITION-SW-COMPONENT-
  TYPE" BASE="swc">Pro22</AR:SOFTWARE-COMPOSITION-TREF>
19
20 </AR:ROOT-SW-COMPOSITION-PROTOTYPE>
21 </AR:ROOT-SOFTWARE-COMPOSITIONS>
22 </AR:SYSTEM>
23 </AR:ELEMENTS>
24 </AR:AR-PACKAGE>
25 ...
```

Listing 4.1: An excerpt of AUTOSAR software component description

This is done explicitly, because there may be several software components within the same package and this is the only way to define what the system is. Defining the root in AUTOSAR is optional and it is also not crucial for the mapping process. But for variability extension in this project (see in following sections) there is only one vehicle feature model. Therefore, this should be provided. Instead, the generator in this implementation chooses any of systems in the root level.

However, the essential point here is the estimation of the path. The absolute path in AUTOSAR is build by concatenating all package names (identified by *SHORT-NAME*) from the top to the corresponding element. For instance, the path for the prototype *CM2ARSystem_autosar1* is */autosar/system/HybConS_TE_UC1*. Its unique identifier would be */autosar/system/HybConS_TE_UC1/CM2ARSystem_autosar1* (see Listing 4.1). In this way model elements are stored in the element table.

Resolving References Another challenge in elements capturing is referencing. The references in AUTOSAR can be expressed in two ways: (1) with absolute paths and (2) with relative paths. Absolute paths always start with a slash followed by the root package. They are used as a package unique identifier. The reference from the example above is not expressed by the absolute path. Formulated as absolute path, the reference would be: */autosar/system/compositions/Pro22*. Unfortunately, the package containing composite software components is not visible in Listing 4.1.

Capturing absolute references does not require any post-processing effort. But, the situation with relative references is a little bit different. Like in the file system, the relative references are the “right excerpt” of an absolute reference and they are related to the containing package. An example for a relative reference is given in Listing 4.1 on line 18. To find the target element this path needs to be converted to the absolute path. For the reference calculation the schema element *ReferenceBase* is used. It holds the left part of the absolute path or in the worst case, it contains a reference to some other reference base. The four ways to express such paths (see [AUT09a]) are described in Table 4.1.

Reference base	Default	Base	Description
absolute	true	-	Default reference base in the package. References without explicit defined base attribute are resolved by using this reference base.
absolute	false	-	Reference base used to calculate the path for elements with explicit specified base attribute.
relative	false	yes	Reference base with relative path and explicit specified base for another reference base. It is used as an intermediate step for calculating the absolute path.
relative	false	no	Relative reference base which requires a default reference base for path resolution.

Table 4.1.: Possible definitions of reference base, [AUT09a]

The first column shows the value of the reference, i.e. the left part of an absolute path. Like any other reference, it can be absolute or relative. *Default* attribute marks a package to be default within its container. That means, the default package is a central point for references within the same sub-tree that do not have an explicitly defined base. There is maximum one default package possible in each container (package). The next attribute is a *Base*, which corresponds to the *SHORT-NAME* of the related package. If a reference of the reference base is absolute, this attribute is ignored by the generator.

The essential issue in the implementation is to decide which reference base to choose. Typically, when the generator captures a reference, it searches for a reference base by traversing the upper-level packages, i.e. in bottom-up order. After all reference bases have been collected, one of them is used. This depends on the definition of the reference. If the base attribute is not specified, then the default reference base is used. Otherwise, a package with the name equal to the base attribute “base” is used. The same strategy is applied to nested reference bases (the last two from the table).

The relative reference to the software component type *Pro22* on line 18 in Listing 4.1 would correspond to one of the last three reference bases from Table 4.1. The default reference base is excluded, because of the explicitly defined base attribute. In order to find the target reference base, all nested reference bases need to be resolved (if existing). A matching reference is the one with the attribute *LONG-LABEL* equals to the “base” at-

4. Implementation of the Prototype

tribute of the software component *Pro22* with the value “swc”. This attribute is a unique identifier for the reference bases.

Resolving Model Elements The AUTOSAR XML schema specifies model elements of all system layers. The elements of the VFB used in this project are just an excerpt of the complete schema. Moreover, there are elements from the VFB that are not part of the mapping like data types, behavior, etc (see mapping strategy in Section 3.3.4). To identify candidate elements it is sufficient to capture their schema element names. These identifiers are used by the *AutosarProcessor* (see Figure 3.6) in the first iteration to capture the model. In the second iteration, references are resolved. For reference building, the strategies from the previous section are applied.

4.1.2. Part II: Variability Extension Generator

Variable model assets are identified by capturing their properties corresponding to variation points. Different realizations of variation points in AUTOSAR are described in Section 2.2.2.3. Here, an aggregation pattern is sufficient to make variability in sub-systems and components possible (see Section 3.3.1.1). The main challenge in this sub-process is the generation of variability logic for captured common and variable model assets. Metamodel elements from the EAST-ADL variability package are generated and interrelated to each other by following the specification conceptually described in Section 3.3.5.1. Figure 4.4 shows a simplified representation of the generation process.

The process generates artifact level and vehicle level elements. The second group is used to provide the system configuration to a developer, i.e. vehicle feature model. It is the only interface to the system developer who configures the system.

4.1.2.1. Artifact Level Variability

After the model transformation is completed, a root software component is identified. This is achieved by capturing the reference from the AUTOSAR model like in Listing 4.1 on line 18 or by finding a candidate). The variability extension is created by traversing the components tree in top-down order.

The first activity in the process, shown in Figure 4.4, takes an element, i.e. a software component from the current composite. Then, it needs to be checked if this candidate is an elementary or composite software component. For the first option there is nothing to do except of proving if it varies. If it does, then a construct *VariableElement* is created within a variability extension and referenced to this model asset. Thus, there is no attribute in a model to define some model asset to be variable. Instead, such a referencing manner is used and therefore the independence between model and variability is achieved.

For each composite, the following is created: a public feature model containing features, referencing the content of a composite, and internal binding, specifying the rules how a variable content is affected by selecting features from the public feature model, i.e. configuration decisions. This public feature model is visible only for a composite container, i.e.

for the composite in the next upper level of the hierarchy. Its internal binding defines how this public feature model is configured. In this way, a configuration is propagated over the whole hierarchy as depicted in Figure 3.5. For each feature of a public feature model

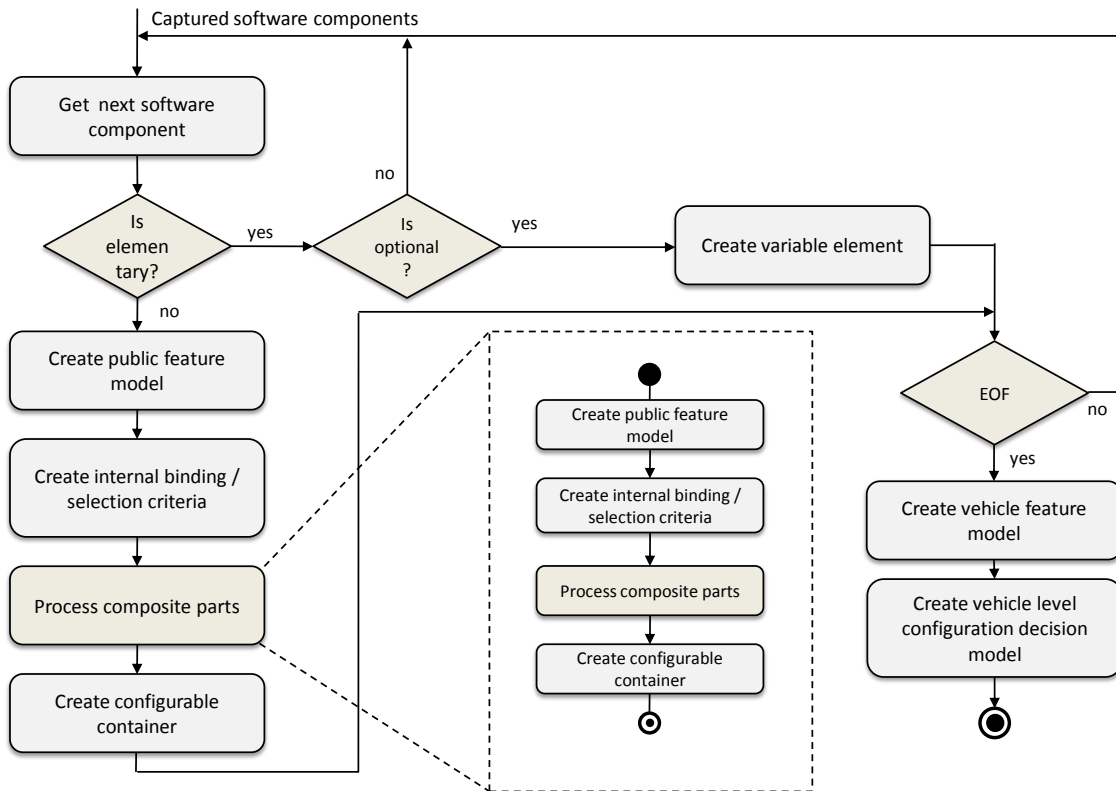


Figure 4.4.: Generation process for variability extension

one configuration decision, according to the concept of transfer function, is created (see Figure 3.6). The parameters *effect* and *criterion* are also estimated. A collection of these configuration decisions, related to one public feature model, is accordingly packaged into one internal binding. Internal binding and public feature model are the most important constructs that hold the whole logic for the system configuration for a specific composite software component.

If a composite contains other composite software components, they need to be processed in the same way. This is done in the activity *Process composite parts*. The processing of a composite software component is done when the variability extension for all its elementary and composite parts is generated. At the end, internal binding and the public feature model need to be packaged into the configurable container. This construct, with the reference to the corresponding composite software component, is created.

4.1.2.2. Vehicle Level Variability

The last two activities of the process are part of vehicle level variability. Here, the vehicle level feature model and a bridge connecting the vehicle level and artifact level in EAST-

4. Implementation of the Prototype

ADL are generated. This is illustrated in the right part of Figure 3.5 (cf. vehicle level and artifact level 1).

The vehicle level feature model is generated by cloning the root technical feature model. The only difference is that features inside the vehicle level feature model are represented by an element *VehicleFeature* which is a specialization of the metamodel element *Feature*. It is extended with attributes like *cardinality*, *isOptional*, etc. The bridge to the artifact level is created by mapping these vehicle features to the features of the root feature model. In fact, they are identical, but the feature model on artifact level is not visible for the configuration.

Multiple Asset Instances In EAST-ADL, a model is build by following the type-prototype approach proposed in the AUTOSAR specification (see component reuse approach in Section 2.2.2.3). Actually, its origin is coming from the type-role approach in UML. The parts of the model are prototypes instantiated from their types. The situation where several prototypes are instantiated from the same type is a case that is specially handled during the generation of vehicle level variability. This is illustrated in Figure 4.5. As mentioned

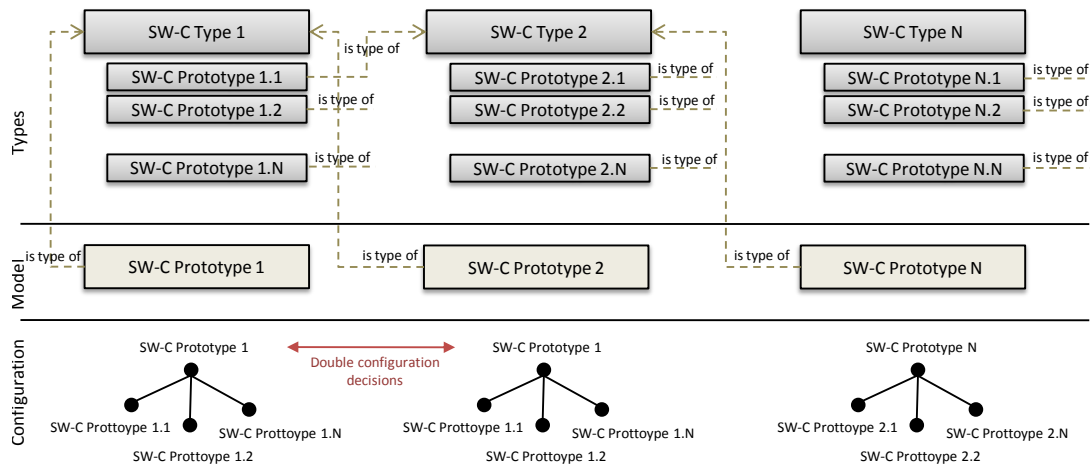


Figure 4.5.: Effect of multiple instances in feature models

in the previous section, for each feature in the model one configuration decision is created. Assuming the situation shown in Figure 4.5 where the prototypes *SW-C Prototype 1* and *SW-C Prototype 2* are instantiated from the same type, i.e. from *SW-C Type 1*, multiple configuration decisions are pointing to the same model assets. The consequence is that a system developer is able to provide different configurations for the same composite, i.e. *SW-C Type 1* in this case. This would be the same as a race condition, and therefore must be avoided. In the generation process for the variability extension the prototype parts are resolved by resolving their types. Therefore, the public feature models of multiple instantiated prototypes would be generated as depicted in Figure 4.5.

To avoid a race condition in the configuration process (see next section) the variability extension is generated only for the first prototype captured by the generator. Thus,

configuring the content of this prototype would affect all other prototypes instantiated from the same type. Most important, the system developer has no ability to configure each prototype individually.

4.2. Application Engineering

4.2.1. Part III: System Configuration

The system configuration can be divided into two sub-processes: (1) generation of system configuration and (2) system derivation. The first process generates the necessary EAST-ADL metamodel elements related to the configuration part of the variability package. The system derivation, on the other hand, builds the system in correspondence to the first process.

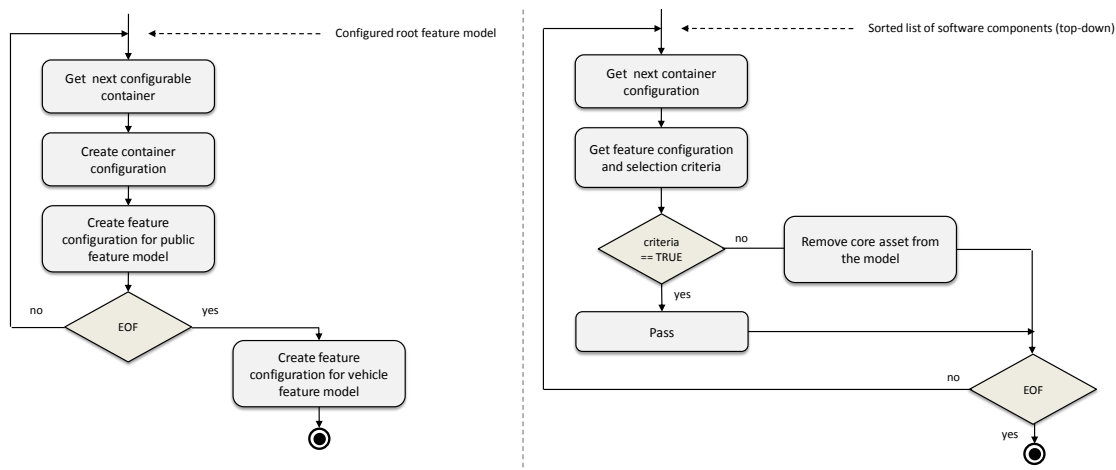


Figure 4.6.: System configuration process: configuration (left) and derivation (right)

System configuration starts with collecting configurable containers generated in the mapping process (see previous section). From each configurable container the configuration decisions are extracted and evaluated. The evaluation process takes the configured vehicle feature model and decides for each configuration decision, which feature belongs to the new generated feature configuration. Feature configuration elements (called *FeatureConfiguration*) denote, which features have passed the evaluation. In addition, the same is proposed for model assets by using *ContainerConfiguration*.

In the derivation process the intersection of two ranges of configuration decisions is made. One group consists of configuration decisions that have passed the evaluation and the other group contains all decisions from the model. The intersection of these groups gives a new group containing configuration decisions that are excluded from the configured system. The related model assets are simply removed from the model. Here it is important to mention that dependent parts of such assets are indirectly removed. For instance, connectors and ports of a variable software component are automatically excluded from the

4. Implementation of the Prototype

model. This could be explicitly modeled by using one of the two mentioned dependency strategies, but it is only necessary for a model validation, i.e. to ensure that concerned ports are optional or at least implicit optional. This is further discussed in Section 7.

4.2.1.1. VSL Expression Evaluator

The crucial factor in the configuration process is the evaluation of collected configuration decisions. As mentioned before, there are many possibilities to express dependencies between referenced and referred feature models. Thus, in order to provide some common solution for these dependencies a decision definition as depicted in Figure 3.6, is followed. For this purposes an interface *VSLEExpressionEvaluator* has to be implemented. It is shown in Listing 4.2.

```
1  /**
2  * This is a part of the HybConS project.
3  *
4  * @author T1000
5  *
6  * Mixed string expression evaluator for variability specification language.
7  *
8  * Evaluates criteria defined in configurable container
9  * (see EAST-ADL specification, section ConfigurableContainer).
10 *
11 * Container's effect allows following forms of the value:
12 * 1 PFM_<public-feature-model-name>#<feature-name>
13 * 2 <artifact-name>
14 *
15 * In the case of multiple occurrences, elements are
16 * separated by comma.
17 */
18 public interface VSLEExpressionEvaluator {
19
20     /**
21     * Evaluates mixed string expression referred to the source
22     * feature model, i.e. to selected features.
23     *
24     * @param configurationDecision container for criteria describing
25     *       combination of features for which the
26     *       condition returns true. E.g. for expression (F1 and not F2)
27     *       the configured feature model must not contain the feature F2
28     *       if the feature F1 is selected.
29     * @param SelectedFeatures selected features of the source feature
30     *       model.
31     * @return true | false
32     */
33     public boolean evaluate(Object configurationDecision, Object selectedFeatures);
34 }
```

Listing 4.2: VSL expression evaluator for variability extension

It takes a single configuration decision and the whole feature model, i.e. selected features, as an input and evaluates the criterion within this decision. Currently, there is a simple implementation class of this interface which is able to evaluate only one variable element in the criterion (cf. configuration decisions on the Figure 3.5). This interface is a possible extension point for the variability mechanism in HybConS architecture (e.g. if a more sophisticated criterion definition is required).

4.3. Integration into the HybConS Tool Environment

Integration into an existing tool environment means on the one hand to provide the generated vehicle level feature model to pure::variants for building corresponding feature and family models and on the other hand to edit the EAST-ADL model with Papyrus. The second does not require any implementation effort, but just provision of necessary Papyrus plug-ins for EAST-ADL inside a tool environment. Both interfaces are conceptually depicted in Figure 4.1 (cf. Eclipse interfaces).

The process of generation is defined as follows: first, a pure::variants feature model corresponding to the EAST-ADL vehicle feature model is generated. Additionally, the family model is generated by cloning the pure::variants feature model. The idea is to express the content of the family model in form of relations to the real EAST-ADL model assets. Unfortunately, pure::variants is not able to access the parts of the EAST-ADL model, i.e. there is no possibility to relate items of the family model to the model assets in FDA directly. For this purpose, the Technical University of Dresden has provided an approach to support the mapping between features and an Ecore-based solution space (see [HKW08]). The implementation of this mapping is available as an Eclipse plugin called *FeatureMapper*, which also includes the functionality for the integration in pure::variants. This would be a solution for the system configuration directly from pure::variants. Unfortunately, the lack of documentation makes it difficult to find out if and where the extension point of this plug-in is and how it can be used by the architecture plug-in, implemented in this work. Therefore, the connection of features and EAST-ADL model elements using the *FeatureMapper* is future work. Alternatively, a solution for the system configuration by using the configuration functionality described in Section 4.2.1 is proposed. This is depicted in Figure 4.7.

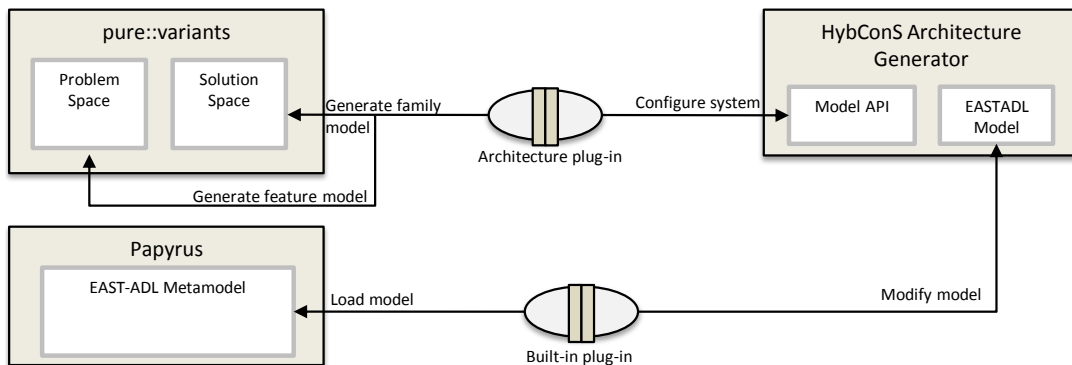


Figure 4.7.: Architecture plug-in: integration in a tool environment

The architecture plug-in uses the API provided by the *HybConS Architecture Generator* to get the vehicle feature model. In correspondence to this model it generates pure::variants feature and family models. It is now up to the system developer to create a configuration, i.e. a variant description model and to derivate the system. The result of a derivation is the variant result model which is used as an input by the API for system configuration.

4. *Implementation of the Prototype*

This plug-in is just a proof of concept and need to be improved in further development.

4.4. Technology

The architecture block introduced in Figure 3.1 is implemented in Java. As development environment Eclipse 3.6.0 (Helios) has been chosen. Concerning the implementation the Spring Framework 3.0.4 for lifecycle management of application components (beans) is used. It allows the replacement of important beans (see Figure 4.2) without much effort. Besides the EAST-ADL metamodel, the EMF-based UML 2.1 (3.0.0) API and metamodel, as well as the metamodel of SysML (SysML specification V1.0 (formal/2007-09-01)) are used. They are also configurable by spring application context depicted in Figure 4.2.

The integration in the HybConS tool environment has been realized in form of an Eclipse plug-in. The architecture plug-in is configured for Eclipse version 3.6.0.

5. Evaluation

Before going in the details with the evaluation, it is advisable to describe the main steps in the process from the user perspective. Basically, an AUTOSAR file (`*.arxml`) describing the software on the implementation level is the input to the transformation engine. To engage the transformation, the user has to import the AUTOSAR file by providing its path as the parameter to the main class *TransformationEngineApp*. An alternative to this would be the use of the simple GUI depicted in Figure 5.4. In addition, the user has to specify whether the variability extension should be generated or not. The result of the transformation is the EAST-ADL model saved as `east.uml`. If the user requires the variability extension, the transformation engine generates it with respect to the EAST-ADL specification. In this case, the user has to configure the generated feature model. To do this, the feature model has to be retrieved from the EAST-ADL model first. Thus, the simple GUI can be used to read the feature model and to derive the variability-free EAST-ADL model.

5.1. Methodology

High reusability and scalability of core assets are the main expectations on this project. Moreover, development costs, quality and time-to-market as primary goals of the SPL are also directly influenced. The aim of this evaluation is to see how the implemented prototype performs depending on these expectations. To do this, several measurement techniques need to be applied. However, because most of the attributes such as time-to-market, quality, etc. are difficult to estimate, the scope of this experiment is focused on a time measurement from which e.g. development costs may be derived. In dependence to provided results, assumptions about remaining goals can be made.

The procedure of development costs estimation is decomposed into the following tasks:

1. Implementation of use-cases (application software in AUTOSAR)
2. Time measuring for single system development
3. Defining requirements for the HybConS architecture platform
4. System development with the prototype (configured EAST-ADL system - FDA)
5. Time measuring for the SPL

After the time measuring for the single system development, the FDA platform of the AUTOSAR model need to be created. The purpose of defining new requirements on the platform is the configuration of different systems.

5. Evaluation

The inspiration for measuring development costs is the motivation curve of the SPL is depicted in Figure 2.1 (left). To compare involved approaches following parameters are required: time to develop the SPL engineering processes, time to develop the single system, time to realize requirements that could not be satisfied by the platform and time to configure the system. The sum of the last two variables gives a Δ (delta) (see Section 2.1.1.3).

At this point the only known value is the time spent to develop the SPL engineering processes which is rounded to 612 hours. Here it is also important to mention that no additional effort is required for this initialization time, because the platform (FDA model) is generated from the existing implementation.

In the following, the evaluation use-cases are introduced.

5.2. Use-Cases

5.2.1. Use-Case1: Simple Vehicle Hybrid System

Figure 5.1 shows the first use case which is provided by the Virtual Vehicle Competence Center (ViF) for evaluation purposes. It is a simple hybrid system enabling the driver to read the state of the battery pack on his display (e.g. on the board). The software system is contained of the body, the hybrid controller and the environment subsystems.

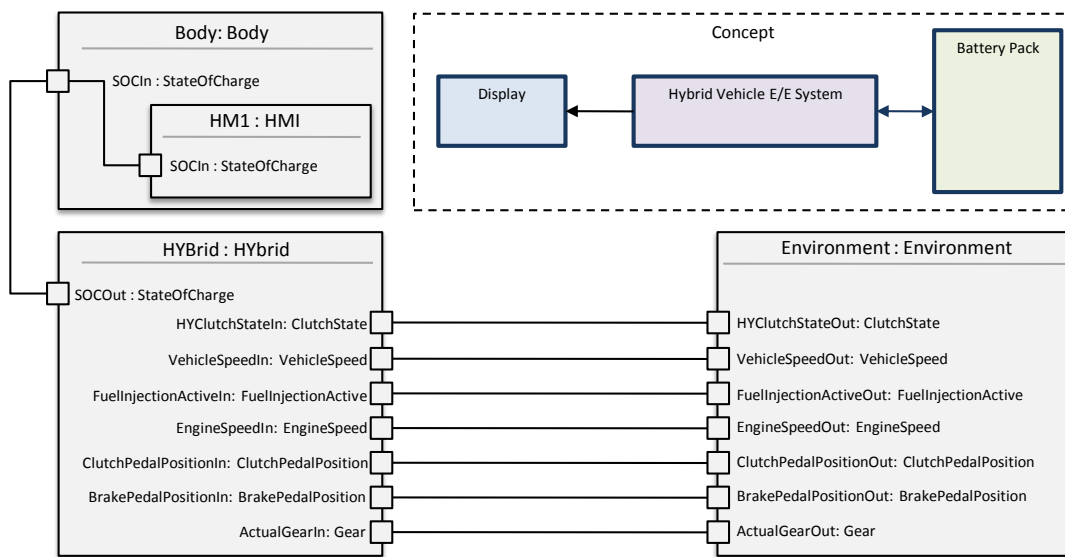


Figure 5.1.: Evaluation use case 1: read battery charge status in a simple hybrid vehicle system

The body subsystem is a main interface between the embedded system of a vehicle and the user (driver, passenger). It includes all the sensing functionality delegated by an user such as seat adjustment, lights system, horn control, etc. However, in this example it provides an HMI (Human Machine Interface) only, which forwards the status value of the battery charge to a display.

HYBrid (hybrid controller in this example) acts as an intermediate layer to the environment. Typically, body and comfort systems do not have a direct access to the environment abstraction layer, because some signals have to satisfy (hard) real-time requirements that for the HMI are not really important (e.g. 1 second latency is no problem). The detailed decomposition of the *HYBrid* is depicted in Figure A.2(a) in the appendix. Its main component is the *HybridControlUnit1*, which controls (adjusts) the signals from the body and the environment. To get the charge value from the environment it requests the BMS (Battery Management System) which contains a single battery pack (*BatteryPack1*) and several modul controllers. Typically, these battery packs are combined from several cells each having its own model controller for state management (e.g. temperature, charge value, etc.). As illustrated in Figure A.2(a), the overall charge value is estimated from all six module controllers.

The environment subsystem contains the software representation of mechanical parts of the vehicle. Here, the charge value is captured by the *PowerTrainControllUnit1*.

5.2.2. Requirements on SPL

The HybConS architecture platform shall provide the hybrid system as illustrated in Figure 5.1. Furthermore, in order to introduce variants, it shall offer the following optional functions:

- (soft real-time): Ability to indicate a driver about remaining drive (e.g. in km) with respect to the current battery charge value.
- (hard real-time): BMS should immediately switch-off the battery pack in the case of a crash.

Furthermore, it is assumed that the platform already contains the system, because its description in AUTOSAR can be easily transformed into EAST-ADL. The only effort here is to extend the platform for the functionality above and to make it configurable. The resulting platform is documented in Section A.4 (note that only the most important parts of the system are shown).

5.2.3. Solution

Because both requirements have an impact on all subsystems, two options to realize the platform are proposed: (1) model additional functions as optional and (2) put variation points to affected subsystems. The last solution is feasible by the implemented prototype, but since all subsystems are affected, the modeling would take more time than in the first case. Namely, here the functions need to be added as optional to the composite types and there is no need for variation points. This option is used.

Concerning technical realization, the first function is realized inside of *HybridControlUnit1* and the value is just forwarded to the *HMI* over *DRVC1* (see Section A.4). The crash detection, on other side, is realized as a part of the BMS (*CrashC1*) which in the

5. Evaluation

case of a crash delivered by the environment component *CD1* sends the signal to the *PowerTrainControllUnit1* to switch the battery off.

5.2.4. Use-Case2: Seat Adjustment in Vehicle System

The purpose of use case 1 was to show how the handling of variations in elementary software components and their ports is realized. In this example the focus is on the configuration of seat topologies in a vehicle. The seat adjustment subsystem handled here is a part of the body and comfort system on the VFB level taken from [AUT09c]. The purpose of the seat adjustment in this example is to control 16 motors distributed over various axes inside of a single seat. This is illustrated in Figure 5.2. Currently, the AUTOSAR model contains three instances of the seat on the front, the middle and the rear line of a vehicle.

The application software of the seat subsystem in interaction with other vehicle subsystems is depicted in Figure A.1(b). The main function here is *SeatAdjMgr* which in correspondence to captured (position) commands from the outside (e.g. by a driver over HMI, Central Locking System or battery status over Inter-Domain Interfaces) controls the seat axes. Note that the red marked attributes in the figure indicate multiple instances of the corresponding type.

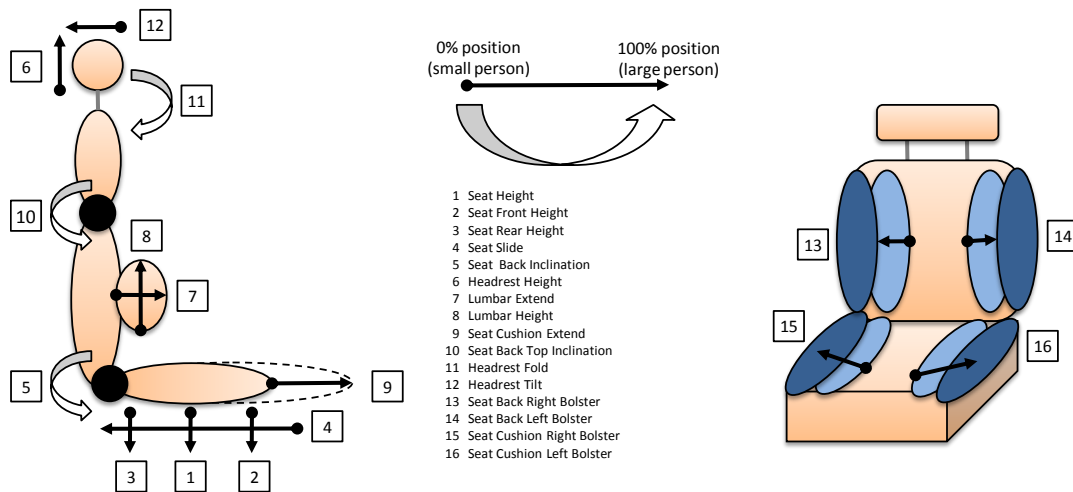


Figure 5.2.: Evaluation use case 2: seat adjustment, [AUT09c]

5.2.5. Requirements on SPL

The HybConS architecture platform shall provide the configuration of the seat topology in such a way that instead of nine seats just four (driver and passenger on front and two other on rear) are present. Furthermore, for the basic class of vehicles the seat subsystem shall provide seats with configuration 1, 2, 3, 4, 5, 9 and 10 depending on the axis types from Figure 5.2. Unlike this, the extended version shall provide all 16 axes of the seat.

5.2.6. Solution

In this use case the system does not need to be extended by a new functionality, but just for variant management. To provide the configuration in a topology, several seat instances are set to be optional. The same is done for axis instances to configure a single seat. Furthermore, it is required to set affected ports of the seat subsystem and the external interfaces as optional too. This has also consequences on the functionality outside of the seat, but here, the seat adjustment subsystem is configured individually and therefore the configuration of external functions is not necessary.

Concerning vehicle classes (basic, extended) such abstract features can not be generated by the prototype, because the top level feature model is build from the system structure. However, the configuration is possible, but in other abstraction, e.g. instead of the basic class, the system engineer has to choose which seats and which axes are a part of the final system.

5.3. Results

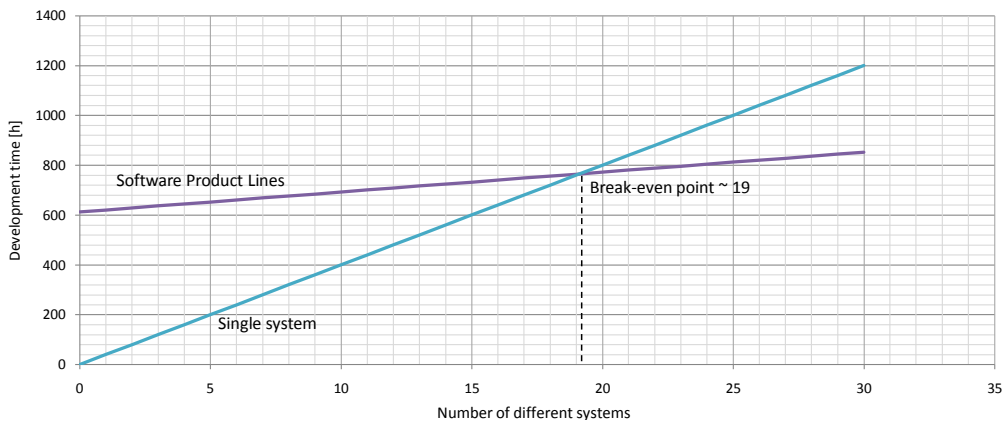
At this point all tasks from Section 5.1 are performed. Concerning the single system implementation of the first use-case the team including a single person spent approximately one week for its realization (≈ 40 hours). Furthermore, the realization of the change request given in Section 5.2.2 (task 3) together with the system configuration (task 4) took about 8 hours. These information are sufficient to construct the development curve of the SPL, if it can be assumed that the implementation of each new product having the same complexity (e.g. number of functions) requires the same time. This is illustrated in Figure 5.3(a). Compared to the *ideal* curve, the break-even point is reached after 19 implementations. In cases like this the applicability of the SPL should be precisely considered before starting with the realization. But fortunately, the first use-case is a small system consisting of about 100 functions only (types and instances) and therefore such results are expected. The platform covers a larger domain and this shows the another use-case.

The seat adjustment subsystem (use-case 2) is a part of the body and comfort system provided by AUTOSAR and therefore no information about the effort for realization are provided. However, this may be approximated by using a data from the first use-case. For this calculation a number of realized functions may be used. Thus, it follows that for the realization of 611 functions of the seat adjustment model approximately 260 hours are required. In addition, to satisfy requirements given in the Section 5.2.5 the model is extended to provide variants in seat and axis topology. For this extension about 8 additional hours are invested. The results of development costs for the use case 2 are shown in Figure 5.3(b). Now, the break-even point is achieved at the third implementation and shows satisfying results. However, this should not be a guarantee for a reliable curve progression. Namely, except of the effort to realize the SPL and a single system the break-even point depends also on *delta* which is the most unpredictable value in SPL. The platform, on the other side, is not focused on a *small* domain, i.e. the contained software architectures may show *any* form. The possible way to hold *delta* on minimum is to have a platform with a large product scope. But this, unfortunately, introduces an additional effort in the

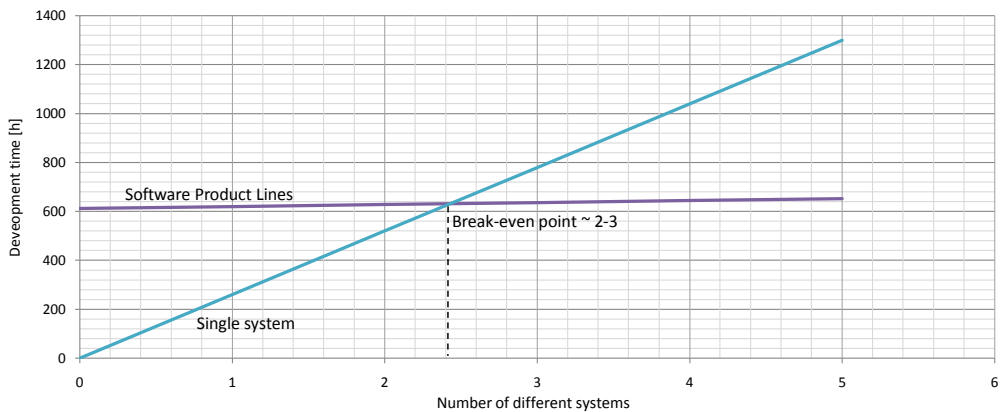
5. Evaluation

variability modeling and also an effort to handle such wide configurations.

The stability of the break-even point is typically increased proportionally to the learning phase of the system, i.e. while developing applications, because the platform is extended for a new functionality by request of *delta* (if possible). The platform in the introduced use-cases is just a single system with few variation points and therefore the results from Figure 5.3(b) can be expected. But however, compared to the single system development, the applicability of the SPL for the software architecture seems to be optimistic.



(a) Use case 1



(b) Use case 2

Figure 5.3.: Evaluation results: development costs for SPL and single systems

Concerning the time-to-market no assumptions can be made, because the generated model is just a part of the system used for documentation purposes. This can be probably documented by release of the HybConS. Anyway, the reduction of the development time shown above improves the time-to-market.

5.3.1. Reusability

Software core assets handled in this prototype are EAST-ADL architectural constructs from the FDA. The implemented variant management mechanism follows the specification of the EAST-ADL. Besides it is not fully implemented, this mechanism offers a system configuration for prototypes of subsystems, software components (functions), ports and connectors. These prototypes are artefacts used to allow differentiation in products.

The seat adjustment model in the use-case 2 is contained of about 600 software components (prototypes). For variants *basic* and *extended* several subsystems, functions and ports are made optional whereby the remaining functionality is reused in both generated models. Without the ability to make ports and connectors optional, their containers would have to be defined as variants and subsequently this would reduce the reusability. Therefore, a degree of the reusability strongly depends on artifact granularity covered by the implemented variant management. For instance, if not supported variant such as variation in attributes is required, the whole container (e.g. port interface) of the affected attribute needs to be switched in order to change the value.

In addition, reusing (evaluated and verified) core assets systematically increases the quality of the software in contrast to the single system development.

5.3.2. Scalability

Scalability in this context denotes the ability to extend and add new functions to the system (platform).

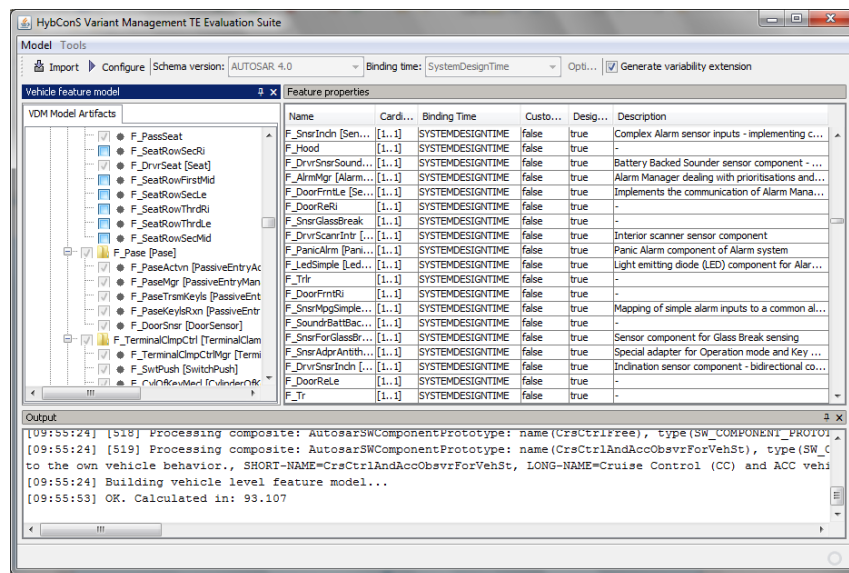


Figure 5.4.: Simple GUI for prototype evaluation

Related to the prototype, this affects the model transformation as well as the variability extension. The architectural platform in this prototype is generated from the software

5. Evaluation

implementation and therefore all new functions are instantly defined in AUTOSAR. If a new function is not supported by the mapping process, the extension guide in Section A.2.2.1 has to be followed. A similar scenario is required to extend the variants (see Section A.2.2.2). However, variant management in this prototype is not as flexible like as the mapping process.

5.3.3. Prototype Evaluation

The two main parts of this prototype are the mapping process, implementing the model transformation (AUTOSAR to EAST-ADL), and the variability extension. The first process is evaluated by creating diagrams of the generated model in MagicDraw and comparing them to the AUTOSAR model. Most of the models have been provided by the Virtual Vehicle Competence Center. The another process is evaluated by configuring different topologies of subsystems, elementary software components, ports and connectors. Again here, the system is evaluated by using MagicDraw. For an easy use of the prototype, a simple GUI is created.

5.3.4. Performance Analysis

The evaluation results described in Section 5 are a kind of proof that the system is working, but another interesting aspect is how well it works. Especially for such model transformations it is interesting to show how the system responds in correspondence to different distribution of elements inside an input model. The aim of this analysis is on the one hand to provide information about efficiency and on the other hand to find bottleneck points for further development.

		Model size [# model elements]						
	Types	9	34	59	109	209	409	310
	Prototypes	6	56	106	206	406	806	518
	Ports	12	112	212	412	812	1612	5410
	Connectors	14	114	214	414	814	1614	4806
	File size [KB]	29	136	242	456	882	1736	12783
Process		Runtime [s]						
Mapping	SAX Parser	0,143	0,219	0,271	0,404	0,671	1,291	3.950
	Model Builder	0,039	0,039	0,064	0,066	0,078	0,108	0,874
	Transformation	1,111	1,744	2,168	3,504	6,133	10,977	29,345
	Summary	1,293	2,002	2,503	3,974	6,882	12,376	34,169
	Variability	0,264	1,561	2,301	5,430	16,823	76,544	69,535
	Configuration	0,072	1,318	3,371	12,657	50,899	216,827	115,367

Table 5.1.: Runtime

Measuring performance is performed by measuring the runtime of the mapping and configuration process. It is also important to define what is expected as an input for model transformation, i.e. what is a typical number of contained software components. For this experiment a model with 800 software components is assumed to be “the worst case model”

(cf. to a typical number of software functions in the vehicle system nowadays, [SZ10]). Tests are performed for 15, 90, 50, 165, 315, 615, 1215 and 828 software components and the response time of the most important methods has been measured. Results are shown in Table 5.1.

The response time for types and prototypes is estimated separately, because prototypes are typically generated faster than types, especially in cases when types are defined as composite software components. Figure 5.5 shows the data from Table 5.1 graphically. The y-axis corresponds to the response time and the x-axis is a test number, i.e. in the same order as defined in the table.

The first six test models are created by a simple generator which creates types and their instances. Types are composite software components consisting of a sub-system hierarchy with a depth of three. The last model (last column) is a “real” one provided as an exemplary model by AUTOSAR. It consists of about 800 components, more than 4000 ports and more than 5000 connectors. The example describes the application software of the power train. This in combination with generated models should result in interesting information about the behavior of the system, because on the one hand there is a linear grow up of types, prototypes, connectors and ports that should indicate efficiency and on the other hand there is a transition between the first six test cases and the last one that should show differences between several processes (mapping process is dependent on more parameters than other processes). The bold marked line ($y = 50s$) indicates

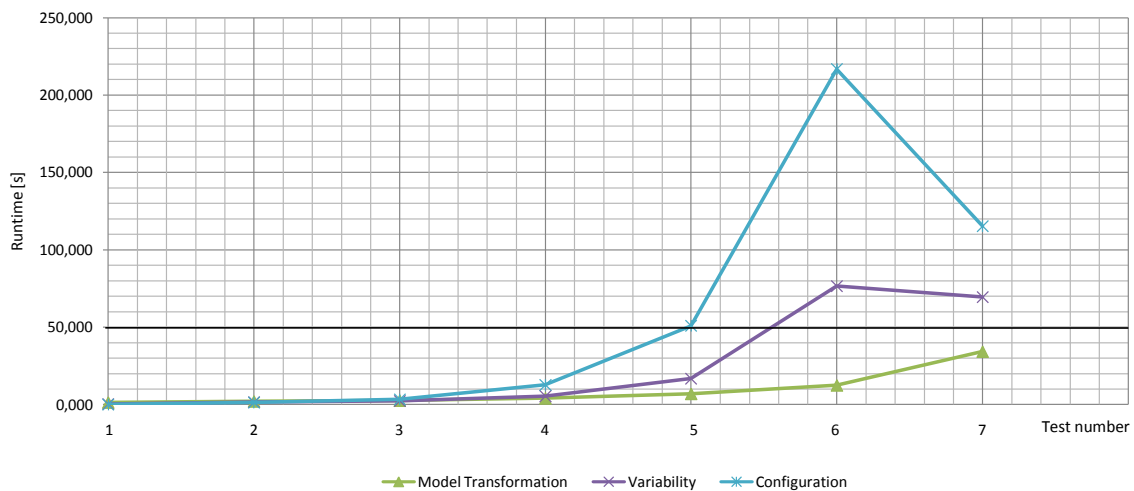


Figure 5.5.: Runtime

the upper bound of the response time the process should not exceed (in this project) in order to be usable compared to a typical prototype. As the figure indicates, this is not the case with the configuration and variability process. They are potential candidates for improvement and should be used for model ranges up to the fifth serie from Table 5.1 only, i.e. containing not more than 600 software components. The reason for such growth is the compositional variability which for each subsystem (composite software component) requires one public feature model (time-critical method is *createVariability(...)*). Thus,

5. Evaluation

the mapping process shows satisfying results, but there are also some bottleneck points in the transformation process which could save at most 15-30% of the time, i.e. 6-10 additional seconds taking the last test case into account. Further reduction of response time would not be possible (at least with this architecture), because the creation of UML model elements takes in average 2 milliseconds and for 11044 elements (last test case) it would be fixed 22,088 seconds.

Another interesting point is the difference in behavior of the considered processes. As depicted in Figure 5.5 a transition between test case 6 and 7 shows different curve progression for the mapping and the two other processes. The mapping process captures all AUTOSAR model elements, but just an excerpt of them are used for the mapping. Therefore, independent of the fact that test case 6 is combined from 1215 components (409 and 806 types and prototypes respectively) the response time of test case 7, is higher because the whole content of 12883 KB need to be processed whereas the configuration and variability extension generator works with software components only.

All processes are sequentially executed by a *TransformationEngine* bean (*cf.* *TransformationEngineImpl* if an improvement is required). For the estimation of response times the following configuration is used: CPU Genuine Intel® T2400, 1.83GHz, 2.00 GB RAM, Windows 7 32-bit.

6. Conclusion

6.1. General

The engineering processes of the SPL are realized in the scope of the practical work of this thesis. The results of the evaluation (see Section 5) show that the application of the SPL for development of the application software from the architectural view is more efficient than developing each variant individually, at least for supported features in the prototype (see Table 6.1). However, it is still too early to make any assumptions about applicability of such an approach for the whole system. As stated in Section 5 the size of *deltas* (effort

Artifact Level: granularity of variants in FDA	Supported
Subsystems	✓
Software components	✓
Ports	✓
Connectors	✓
Vehicle Level: feature relations	
Optional	✓
Mandatory	✓
Or (OR)	
Alternative (XOR)	
Inter-feature relations (exclude, include)	
Configuration	
Binding times	✓

Table 6.1.: Realized prototype features with respect to variant management (granularity is constrained by *VariableElement*, [Con10a])

for development with the SPL) is strongly dependent on the product scope behind the platform and the efficiency of the variant management mechanism. In this prototype, the platform is generated from the software implementation which subsequently together with the variant management influences the development costs line from Figure 5.3. Therefore, an evaluation with more reliable results may be expected in later development iterations of the HybConS project.

The mapping process, which translates the implementation platform to the EAST-ADL FDA, is realized as a model-to-model transformation that can be classified as a direct manipulation approach as introduced in Section 2.3.1.2. Such imperative (contrary to declarative) realisations are typically intended for a single domain (stable metamodels), but they, on other side, lack in flexibility and have no direct support for bi-directionality.

6. Conclusion

However, this approach has been chosen, because of the nature of the AUTOSAR XML schema. In addition, the implemented EAST-ADL API is intended for further use outside the transformation (e.g. to build the traces between abstractions).

The variability extension is implemented with respect to the EAST-ADL specification 2.1, [Con10a]. Such compositional variability in contrast to the global variant management is a more practicable approach concerning the OEM-supplier relation, i.e. for the whole V-Model (see Section 2.1.2.2). In addition, the implementation of the configuration logic within the model allows the system engineer to work with the platform by using any EAST-ADL tool. Furthermore, this makes the product line portable. For instance, the system model (e.g. XMI) may be exchanged between developers responsible for the configuration of different ECUs. The prototype supports almost the whole variability extension, but there is still some work to do, especially in criteria evaluation (see Section A.7).

To sum it up, the implemented prototype provides the SPL engineering processes supporting the variant management for the automotive software architectures. The platform is built in the scope of the model mapping between AUTOSAR and EAST-ADL, whereby the whole variant management is derived from the EAST-ADL specification. The variable elements are captured with respect to the variant handling in AUTOSAR and mapped to the compositional variability. The information about variants (AUTOSAR variability matrix, system conditions, etc.) are stored in features and can be handled by the client. The prototype supports differentiation in subsystems, software components, ports and connectors (see Section 5). This allows to configure topologies, subsystems and individual software components with respect to their ports and connectors (e.g. variation in datatypes of port interfaces is not supported). From the HybConS project perspective, this functionality can be used to configure the coarse grained variability.

6.2. External vs. Internal Variability

As stated in Section 2.1.1.2 the feature model on the top level can also provide external variability which in addition may be present in lower levels of abstractions. Figure 6.1 shows the amount of external and internal variability in this project.

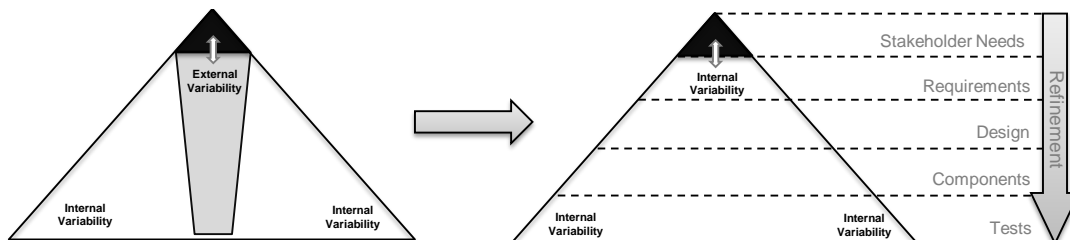


Figure 6.1.: Variability pyramid in HybConS architecture: ideal (left) and real (right) amount of external and internal variability

The situation on the right side of the figure compared to the *ideal* variability pyramid

differs in absence of external (customer visible, abstract) variability. The reason for this is a lack of information in the generation process of the variability extension. The only information used to build the configuration models with respect to compositional variability is the generated architecture. However, the configuration of the system is still possible, but confronted with too much technical details. To reduce the complexity of the top level feature model, the analysis level as an abstraction of the architecture has to be built.

6.3. Plain Propagation

The generation of the compositional variability is performed with respect to the plain propagation pattern (see Section 2.2.2.5). The current implementation of the prototype supports the evaluation of a single link between features in the FM in the configuration decision. This is satisfying for the generated variability, but however, more advanced VSL expressions would not be understandable by the evaluator. This handicaps the full support for EAST-ADL variant management.

7. Future Work

7.1. Mapping Process

7.1.1. Model Transformation

The detailed mapping and the implementation status are given in Section A.3. They show that the already implemented model specification covers just an excerpt of both meta-models, i.e. 13,89% for AUTOSAR VFB and 64,10% for EAST-ADL FDA respectively. For the proof-of-concept implementation, the most important metaclasses are mapped, but in later development, the remaining metaclasses may be required. Very probably not the whole specification of the AUTOSAR VFB will be used, but anyway it has to be completed. Thus, the current implementation allows to add new metamodel elements without big effort. For this purpose, several extension guide examples are shown in Section A.2.

Another subject for improvement is the model consistency. Any changes in the architecture should affect all other related domain assets (implementation, tests, etc.) and vice versa. Without the ability to perform the inverse transformation, i.e. towards AUTOSAR VFB, this could be satisfied if changes occur in the implementation block only. Therefore, a transformation from EAST-ADL FDA to AUTOSAR VFB is necessary.

Optionally, but not irrelevant, the model transformation should provide a detailed report to the client, especially information on the warning and error level. This requires the design and implementation of an adequate error model. For instance, it should distinguish between failures caused by the system and failures caused by a user. This would help to fix possible bugs in the system as well as to warn the user about performed malfunction (e.g. an invalid model). The current implementation logs some possible failures in a file (e.g. missing references between prototype and type). It is also important to provide these information in some structured form. This makes it possible to present them in an external external tool environment.

The last suggestion is the validation of the model against the AUTOSAR XML schema. It should ensure that the transformation is working with a valid model. However, this would have a meaningful penalty on performance, since the schema file is more than 2MB and the size of a model could be in a range of several megabytes (see Section 5.3.4).

7.1.2. Behavior Mapping

Since the EAST-ADL extension provides the ability to describe the behavior of the structure, it would be meaningful to include AUTOSAR behavioral units, i.e. *runnables* in the mapping process. However, the description of the behavioral semantics in AUTOSAR may be complex and very probably not all parts of the internal behavior would have a

7. Future Work

corresponding candidate in the EAST-ADL model, but basic mapping would be possible. More useful information on this suggestion are given in [CFJ⁺08].

7.2. Variability Extension

7.2.1. External System Configuration

The architecture plug-in depicted in Figure 4.7 uses an internal mechanism to configure the system (cf. Figure 3.5). Inside this mechanism the features are traced to the model assets. A more flexible solution would be to realize these traces in `pure::variants`. This would allow the system developer to modify the reference architecture without changing rules specified in EAST-ADL. As mentioned in Section 4.3 this could be realized by extending the existing architecture plug-in to use the *FeatureMapper* in order to trace the features to model assets. This is illustrated in Figure 7.1.

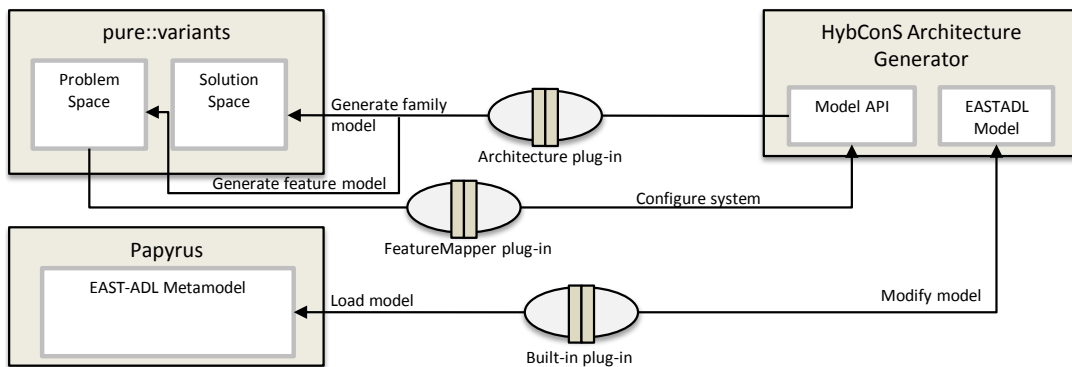


Figure 7.1.: External system configuration with the FeatureMapper plug-in

The difference to Figure 4.7 is the separation of the path between `pure::variants` and EAST-ADL. The architecture plug-in has to be used for the retrieval of vehicle feature models and the generation of family models, but additionally it should map the content of the family model with the model assets in EAST-ADL by using the *FeatureMapper* plug-in. In this way, two mechanisms for system configuration would exist in parallel. The EAST-ADL mechanism could be used to configure the system outside of the HybConS tool environment (e.g. with some other tool conforming to the EAST-ADL specification).

7.2.2. Formula Expression

A single rule which describes a consequence on feature selection is defined by the criterion parameter in the configuration decision. A formula expression of this rule conforms to the variability specification language (VSL). In this way, a very flexible solution to express rules is achieved. Currently, a simple form of such a rule is implemented (see Figure 3.6). It is enough to bind a single feature with a single model element or a feature, but if more sophisticated expressions are required the implementation of the *VSLExpressionEvaluator* needs to be extended. To fully support this language, an adequate parser need to be

implemented. A possible solution would be the implementation of the language grammar (e.g. with ANTLR, like in AUTOSAR) and the generation of the parser and lexer.

7.2.3. Attributes, Associations and Property Sets

The aggregation pattern introduced in Section 2.2.2.3 allows to handle all three locations that can vary from the VFB perspective [AUT09d]: (1) software components, (2) ports and (3) connectors. In this project, the pattern is realized for software components only. In a similar way, the other two elements should be handled. To allow full system configuration for the HybConS architecture, all three patterns (see Section 2.2.2.3) have to be implemented. Associations and property sets belong to the same kind of variability as aggregation, i.e. variation point decides about existence of an element. Unlike these, the attribute pattern defines the variation of the value. To allow such a configuration, these variation points need to be captured in the vehicle level feature model and the configuration just needs to execute the expression inside the *FormulaByCondition* (expression conforming AUTOSAR grammar).

7.3. Diagram Information

The AUTOSAR XML schema does not hold any diagram information about the underlying model. This diagram information has to be generated in order to provide documentation in a user-friendly form. Currently, the generated EAST-ADL model can be read with any UML tool (supporting EAST-ADL 2.1), but unfortunately it is mainly represented in a tree view. However, some UML tools like MagicDraw allow to generate a diagram for a given UML model, but this generation should also be possible inside the HybConS tool environment.

A. Appendix

A.1. Tool Evaluation Criteria

The criteria attributes for tool rating are taken from the master thesis of Andrea Leitner [Lei09] and Andreas Haselsberger [Has10] and extended by some additional attributes based on [BCD⁺00], [ODF07], [DDN07] and [Sit].

Nr.	Criterion	Definition
Product Line Engineering criteria		
1	Attribute management	<ul style="list-style-type: none">• Differentiate between SPL requirements and product requirements• Manage requirements attributes (identifier, description, justification, cost,...)• Ability to capture future requirements• Ability to capture new requirements during derivation• Autobuild with given specifications (mining)
1	Feature and variability modeling	<ul style="list-style-type: none">• Help to model FODA-like concepts (feature decomposition, feature type, cardinalities, dependency links,...)• Support different abstraction levels• Support global constraints
3	Feature model maturity	<ul style="list-style-type: none">• Allow to define a PL metamodel• The tool should be unambiguous• Support product line evolution

A. Appendix

4	Constraint checking and propagation	<ul style="list-style-type: none"> • Support validation checking for the PL model and metamodel • Check consistency of product model and PL model • Check consistency of model and artefact base • Support constraint propagation • Compare artefacts to a “standard” • Rule-checking
5	Product derivation	<ul style="list-style-type: none"> • Help to derive specific products with guidance and visualization
6	Domain engineering management	<ul style="list-style-type: none"> • Support the creation of domain artefacts • Support the management of domain artefacts • Map domain artefacts to corresponding features • Search functions, to find suitable artefacts
7	Application engineering management	<ul style="list-style-type: none"> • Support the management of application artefacts • Reuse of domain artefacts
8	Repository	<ul style="list-style-type: none"> • Version management of artefacts, documents or possibility to integrate such a tool • Re-create any version of a product

A.1. Tool Evaluation Criteria

9	Model comparison	<ul style="list-style-type: none"> • Compare different products • Compare different versions of a product
Management Criteria		
10	Impact analysis	<ul style="list-style-type: none"> • Perform impact analysis when changing requirements or models • Perform impact analysis when changing interlink requirements
11	Reporting	<ul style="list-style-type: none"> • Ability to generate reports
Technical criteria		
12	Access mode	<ul style="list-style-type: none"> • Allow multi-user access • Allow access with profiles (define the metamodel / use it)
13	Technical environment	<ul style="list-style-type: none"> • Support synchronization • Interoperability: support import and export from other tools (APIs, neutral format files, etc.) •
14	Usability	<ul style="list-style-type: none"> • Intuitive usage • Stability and efficient support • Offer high accessibility of functions, zoom, views, ... • Ability to handle great amount of artefacts

A. Appendix

15	Automatic filters	<ul style="list-style-type: none">• Automatic filters on requirements presentations and report generation
16	Tool configuration	<ul style="list-style-type: none">• The tool should be configurable for specific user needs• Adaption to current organisation
17	Extensibility	<ul style="list-style-type: none">• Should be extensible to integrate existing platforms into the PL
18	Flexibility	<ul style="list-style-type: none">• Changes should be possible at each stage of development (also in derived products)
19	AOB	<ul style="list-style-type: none">• Tool costs and training costs /amortisation time• Light charge of installation, maintenance and migration cost• Vendor stability• Flexible licensing service

Table A.1.: Criteria attributes for tool evaluation and selection

A.2. Extension Guide

In this section some useful information about possible extensions of the existing architecture are provided.

A.2.1. Interfaces

Related to the domain engineering process depicted in Figure 3.1 it is important to know how the architecture block is interfacing with the implementation and the analysis block, because these blocks are potentially sources for change requests in functionality.

Typically the model transformation is engaged by the implementation block by using the interface *TransformationEngineService* with a single method called *transform*. If this is not satisfying, all beans from the application context are accessible to the caller. To access the model directly, the class *EastadlModelAPIImpl* has to be used. It holds the model in memory while the application is running. However, the interfaces implemented by this class, *EastadlMappingManager* and *EastadlVariantManager*, provide mapping and variability functionality and not the primitives to modify (e.g. creation of trace links between model elements) the model. But in a similar way, these primitives can be implemented if required.

The following steps are required to extend the functionality of the interface exposed to the implementation block:

1. Specify the interface describing the functionality to be added (e.g. make traces between implementation and design model elements)
2. Implement low level functionality in the abstract class *EastadlProfileAPI* (e.g. read model elements, add attributes, etc.)
3. Implement the interface from step 1 inside the class *EastadlModelAPIImpl* which acts as a wrapper to the low level API.
4. Adapt the *TransformationEngineService* accordingly to use the new functionality

To extend the variability functionality it is not necessary to add new interfaces, instead just the interface *EastadlVariantManager* needs to be extended. Currently, the class *FeatureDescription* is used to expose the vehicle level to external tools. It provides the feature information described in Table A.2. In the same way any other part of the EAST-ADL model can be exposed.

Class FeatureDescription		
Category	Name	Description
Field Summary	name	Feature name.
	cardinality	Multiplicity [m..n], m>0, n>0, n>=m.
	rawFeature	Real feature object (optional).
	isCustomerVisible	External or internal feature.
	isDesignVariabilityRationale	Decision whether a feature is related to design asset directly.
	attributes	Any external data in form of key/-value pairs.
	isRemoved	Selection status.
	bindingTime	Binding time of a variant.

Table A.2.: API documentation for the class *FeatureDescription*

A. Appendix

An example for the usage of both interfaces (mapping and variability) is given in the class *TransformationEngineWrapper*. It executes the transformation and afterwards retrieves the generated feature model. Finally, the configured feature model is sent to the *EastadlVariantManager* in order to configure the system.

A.2.2. Feature Extension

As concluded in Section 6 just an excerpt of the AUTOSAR VFB specification is used for mapping, i.e. the most important structural parts. Therefore, it is expected that most of the extension activities are related to the implementation of the remaining metamodel elements. This section shows the main steps to extend the implemented prototype features.

A.2.2.1. Adding Model Elements

The mapping process starts with capturing AUTOSAR model elements. For this purpose the abstract class *AutosarSAXParser* is used. It uses the internal stack to build the AUTOSAR paths and the model table (see Section 4.1.1.2). The following steps are required to add a new metamodel element into the mapping process (steps 1-3 correspond to the path from an AUTOSAR model element to the transformation, steps 4 and 5 are the EAST-ADL part and steps 7 and 8 correspond to the mapping):

1. Define constant of an element in the static class *Constants*.
2. Use the class *AutosarSAXParser* to capture the element by following schema:
 - Element start and its attributes are captured in the method *startElement(...)*
 - Element end is captured in the method *endElement(...)*
 - The text content (text node) of an element is captured in the method *characters(...)*

Because an element may be a part of a software component, connector, port or other element, the following containers are available when the method *characters* is triggered: the current software component (member *currentPart*), the current port (*currentPort*), the current connector (*currentConnector*), the current variation point (*currentVariationPoint*) and the reference base (*referenceBase*). Thus, it is required to check whether the captured model element is really part of the required container. If no container is required an element can be simply added to the stack and to the model table.

3. Extend the corresponding processor class to add an element to the intermediate AUTOSAR model. Currently, processor classes for software components, ports, connectors, relations, interfaces and primitives are available. If no adequate processor is present, a new one needs to be implemented by extending the abstract class *AutosarComponentProcessor*. The purpose of these processor classes is to map a schema specific element from the model table to the schema independent model element. Additionally, relations between decoupled elements are resolved (attributes, connectors, parts, etc.).

4. Extend the class *EastadlProfileAPI* to provide the creation of the target element in EAST-ADL corresponding to the captured model element (e.g. create *HardwareComponentType*).
5. Extend the mapping interface *EastadlMappingManager* to map elements.
6. Extend the method *transform* in the *TransformationEngineService* to forward captured model element to the EAST-ADL part of the mapping process.

A.2.2.2. Adding Variants

The current implementation of the EAST-ADL variant management mechanism captures only variants corresponding to the aggregation pattern in AUTOSAR (see 2.2.2.3) only. In a similar way, variants of property sets and associations can be realized. Note that a variation point in AUTOSAR is a function of the binding time, applied pattern and the target element. Therefore, not all elements can be optional. In the intermediate AUTOSAR model (generated after parsing) each generic AUTOSAR element *AutosarNamedElement* has an attribute *variationPoint*, but it does not mean that it can vary. This is constrained by the scope defined by the enumeration *VariationPointScope*. In order to add a new variant the following steps need to be performed:

1. Define the target element in the enumeration *VariationPointScope* (metamodel element that should vary).
2. Set the scope to the new captured variation point inside the method *processVariationPoints* in the class *Autosar4SAXParser*.
3. Create EAST-ADL element *VariableElement* in the class *TransformationEngineImpl* and refer it to a new variant (model element that should vary).
4. In this step the variability logic is created. The feature from the lowest level feature model in the multi-level feature tree should refer to a real EAST-ADL model element to which the *VariableElement* was assigned (step 3). Other occurrences of this feature in upper levels of the tree need to refer to the features only (see Figure 3.5). First a variant container (part that contains this variant) needs to be found. If a container is not supported by this mechanism, it has to be previously implemented by following these steps. Currently, possible containers are composite and elementary software components. Their variability extension is created in the method *buildPublicFeatureModel* of the class *EastadlModelAPIImpl*. Here, the target variable element need to be retrieved as a part of the composite. Then metamodel elements *Feature*, *SelectionCriteria* and *ConfigurationDecision* need to be created in the same way as it is realized for other parts of the container. The feature is then a part of the container's feature model. To do this, follow the schema for defining variants realized in the method *buildPublicFeatureModel* in the class *EastadlModelAPIImpl*.
5. Extend the method *deriveSystem* to capture the new variant.

The attribute pattern differs from the other three (property set, aggregation and association). The variation point related to this pattern does not define the existence of

A. Appendix

the variant, but of its value. The steps for the generation of the variability extension are the same as described above, but the derivation process needs to be extended. Instead of removing a model asset it needs to update its value. This extension can be added to the method *deriveSystem*.

A.2.2.3. Adding AUTOSAR XML Schema

The following steps are required to implement a new schema:

1. Add a new entry to the enumeration *SupportedVersion*.
2. Create a class extending the abstract class *AutosarSAXParser*.
3. Add an option to create the instance in method *newSAXParser* of the class *AutosarProcessorImpl*.
4. In the parser class from the step 1 capture elements required for mapping and put them into the field *content* of the extended class. It is used by the *AutosarModelBuilder* to generate an intermediate model.
5. Add an option to create processor instances in the method *initProcessors* of the class *AutosarModelBuilder*.
6. Create processor classes by implementing the interface *AutosarComponentProcessor*.

A.2.2.4. Adding EAST-ADL Metamodel

The following steps are required to add a new EAST-ADL metamodel to the system:

1. Update affected methods in the abstract class *EastadlProfileAPI* to support new metamodel elements.
2. Update constants in the static class *Constants*.
3. Change a path of the new metamodel in the configuration file *configuration.xml* in the bean *eastadlRegistry*.

A.3. Mapping Details

A.3.1. AUTOSAR

ID	Name	Description
Software component types (* - SwComponentType)		
A1	Application*	Hardware independent software component.
A2	ComplexDeviceDriver*	Hardware dependent software component (it has direct access to the hardware).
A3	EcuAbstraction*	The software component used as an access point to the ECU periphery from the application software.
A4	NvBlock*	The software component used to provide non volatile shared memory that can be accessed by other software components.
A5	Parameter*	The software component used to provide shared parameter space that can be accessed by other software components.
A6	SensorActuator*	Access point to the hardware sensor/actuator from the application software.
A7	ServiceProxy*	The software component used to enable inter-ECU communication.
A8	Service*	The software component used to configure the services on ECU.
A9	Composition*	Composite software component.
Software component prototypes		
A10	SwComponentPrototype	The prototype which can be typed by any of the software component types (A1-A9).
A11	RootSwCompositionPrototype	Prototype typed by a root composition.
Port prototypes		
A12	PPortPrototype	Provider port prototype.
A13	RPortPrototype	Requester port prototype.
Port interfaces		
A14	ClientServerInterface	The interface defining operations between the client and the server.
A15	NvDataInterface	The interface containing the data to be exchanged between software components and shared memory provided by <i>NvBlockSwComponentType</i> .
A16	SenderReceiverInterface	The interface for data-oriented transmission.
Connectors		
A17	AssemblySwConnector	The connector used to connect inner ports only.
A18	DelegationSwConnector	The connector used to connect outer and inner port.
Variability		

A. Appendix

A19	BindingTime	Time to bind the variant. AUTOSAR supports following binding times: SystemDesignTime, CodeGenerationTime, PreCompileTime, LinkTime and PostBuild.
A20	PredefinedVariant	A variant corresponding to the specific configuration.
A21	SwSysCond	System condition defining the conditions under which the variants are bound.
A22	SwSysConstValue	The values of the system constant.
A23	SwSystemConstantValueSet	The container for system constant values.
A24	SwSystemConst	The system constant used to select a particular variant.
A25	VariationPoint	This element enables the corresponding model element to show a variable character.
References		
A26	ContextComponentRef	Reference to a port role.
A27	InnerPortRef	Inner end of delegation connector (not visible from outside).
A28	PackageRef	Reference to a package (used by e.g. the reference base).
A29	OuterPortRef	Outer end of delegation connector (port of the composite exposing functionality to the outside).
A30	ProviderIRef	Reference to provider port inside of a delegation connector.
A31	ProvidedInterfaceTref	Reference to provided interface.
A32	RequesterIRef	Reference to requester port inside of a delegation connector.
A33	RequiredInterfaceTref	Reference to required interface.
A34	SoftwareCompositionRef	Reference to software composition.
A35	SwSystemConstantValueSetRef	Reference to the metaclass SwSystemConstantValueSet.
A36	SysCRef	Reference to the system constant used by the variation point.
A37	SwSystemConstRef	Reference to the system constant used by the metaclass SwSystemConstantValueSet
A38	TargetPPortRef	Reference to provider port.
A39	TargetRPortRef	Reference to requester port.
A40	TypeTref	Common reference (e.g. reference to software component type).
Attributes		
A41	ArPackage	Schema element used for packaging model elements.
A42	Base	Reference to base (absolute or relative path).

A43	Components	The container for components.
A44	Connectors	The container for connectors inside of a particular composite software component.
A45	Desc	A description of model elements.
A46	Dest	Reference to a destination model element (absolute or relative path).
A47	IsDefault	A value that specifies if the reference base holding this attribute is default or not.
A48	L4	Long name of a model element.
A49	L2	Description of intent.
A50	Ports	The container for ports.
A51	ReferenceBase	A model element used to calculate an absolute path.
A52	ShortLabel	The element name that is used as an unique identifier for some model elements (e.g. for reference base).
A53	ShortName	The element name that is used as an unique identifier for all model elements.

Table A.3.: Analysed AUTOSAR VFB metamodel elements

A.3.2. EAST-ADL

ID	Name	Description
E1	EAST-ADL2	Root package in EAST-ADL metamodel containing the following sub-packages used in this project: Structure including FunctionModeling, SystemModeling, FeatureModeling, HardwareModeling and VehicleFeatureModeling, Infrastructure including Datatypes, UserAttributes and Elements, and Variability.
Design function types		
E2	BasicSoftwareFunctionType	Abstraction of middleware functionality.
E3	DesignFunctionType	Common representation of a function in FDA.
E4	HardwareFunctionType	A transfer function of the hardware component in HDA.
E5	LocalDeviceManager	Function used for calibration of values retrieved from the <i>BasicSoftwareFunctionType</i> .
Design function prototypes		
E6	DesignFunctionPrototype	Instance of a metaclass <i>DesignFunctionType</i> .
Ports		

A. Appendix

E7	FunctionClientServerPort	The port for control-oriented communication.
E8	FunctionFlowPort	The port for data-oriented communication.
Interfaces		
E9	EADatatype	The data type of ports in data-oriented communication.
E10	FunctionClientServerInterface	The interface describing operations between client-server ports.
Connectors		
E11	FunctionConnector	The connector used for data exchange between communicating ports.
Variability		
E12	BindingTime	The time at which a variant is bound.
E13	ConfigurableContainer	The container for a public feature model and an internal binding.
E14	ConfigurationDecision	A single decision describing a consequence on feature selection.
E15	ContainerConfiguration	The list of configuration decisions that have evaluated to true.
E16	Feature	System characteristic on high abstraction level.
E17	FeatureConfiguration	The list of features included in the configured system.
E18	FeatureModel	The container for features.
E19	InternalBinding	A collection of rules for the configuration of the container the binding is related to.
E20	PrivateContent	The reference to a content that needs to be invisible (e.g. excluded from the model in the derivation process)
E21	SelectionCriterion	The reference to model elements used in criterion attribute of the metaclass <i>ConfigurationDecision</i> .
E22	VariableElement	Optional model element.
E23	VehicleLevelConfiguration DecisionModel	A collection of configuration decisions which bind the vehicle level feature model with the technical feature model.
E24	VehicleFeature	Features derived from the metaclass <i>Feature</i> to provide functionality for variant management.
Attributes		
E25	BindingTimeKind	Enumeration containing the following binding times: systemDesignTime*, codeGenerationTime*, precompileTime*, linkTime*, postBuild, runtime. * - binding times used in this project.
E26	ClientServerKind	Specification for port roles: server or client.

E27	EAStrng	Common datatype used as a replacement to AUTOSAR data types.
E28	UserAttributeValue	Extension point for EAST-ADL in form of key/-value pairs. Currently, it is used to store description (A45) of AUTOSAR software components.
External elements		
E29	FlowDirection	SysML enumeration with the following direction options for communicating ports: in, out and inout.
E30	PortAndFlows	SysML package used by EAST-ADL to specialize SysML ports for domain specific attributes.

Table A.4.: Analysed EAST-ADL metamodel elements

Note that members of EAST-ADL metaclasses are not included in Table A.4.

A.3.3. Mapping

Source ID	Target ID	Remarks
A1	E3	-
A2	E4	-
A3	E2	-
A4	E3	Mapped to the common function type, because there is no special function type in EAST-ADL representing shared non volatile data.
A5	E3	Mapped to the common function type (reasons are the same as above).
A6	E5	-
A7	E3	Mapped to the common function type (reasons are the same as above).
A8	E3	Mapped to the common function type. Actually, this component is not intended to be used in the system design phase, but it's type may be present there.
A9	E3	<i>DesignFunctionType</i> can be configured to be an elementary or composite function.
A10	E6	-
A11	E6	Additionally, root design function prototype is attached to the FDA as a child element.
A12	E7 or E8	Target element depends on the port interface type. To distinguish between provider and requester port, a direction (in, out, inout) is specified.
A13	E7 or E8	-

A. Appendix

A14	E10	-
A15	E9	Data exchanged between <i>NvSwComponentType</i> and other software components can be specified by <i>EADatatype</i> .
A16	E9	-
A17	E11	Distinguishing between the assembly and delegation connector is automatically done by the mapping process. Thus, in EAST-ADL it is not possible to specify a type of the connector explicitly, but this information can be easily retrieved from the structure by the following constraints on connectors in Section <i>FunctionConnector</i> in [Con10a]. Thus, this is only required for the inverse transformation.
A18	E11	-

Table A.5.: Detailed mapping between elements of AUTOSAR VFB and EAST-ADL FDA (Source ID - AUTOSAR model element id, Target ID - EAST-ADL model element id)

The variability extension is not directly mapped. Instead, it uses the information from A19 to A25 from Table A.3 to generate the corresponding EAST-ADL model elements.

A.3.4. Implementation Status

Tables A.6 and A.7 show the current implementation status expressed in percents. It is calculated by counting implemented and remaining metamodel elements from the AUTOSAR software component description template and the EAST-ADL FDA.

Package name	Implemented [%]	[# metaclasses]	Note
Software components	100,00	9	-
Ports	100,00	2	-
Interfaces	100,00	6	-
Connectors	100,00	2	-
Data types	0,00	15	-
Internal behavior	0,00	37	-
Implementation	0,00	2	-
Variability	33,33	3	Calculated in correspondence to supported features in [AUT09d]
Communication parameters	0,00	16	-
Blueprint	0,00	7	-
End-to-end-protection	0,00	2	-

A.3. Mapping Details

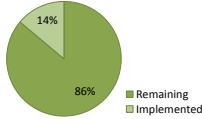
Documentation	0,00	1	-
Measurement	0,00	5	-
Annotations	0,00	15	-
Interface mapping	0,00	22	-
Summary [%]	13,89		

Table A.6.: Implementation status for AUTOSAR part of the mapping process

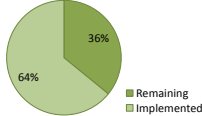
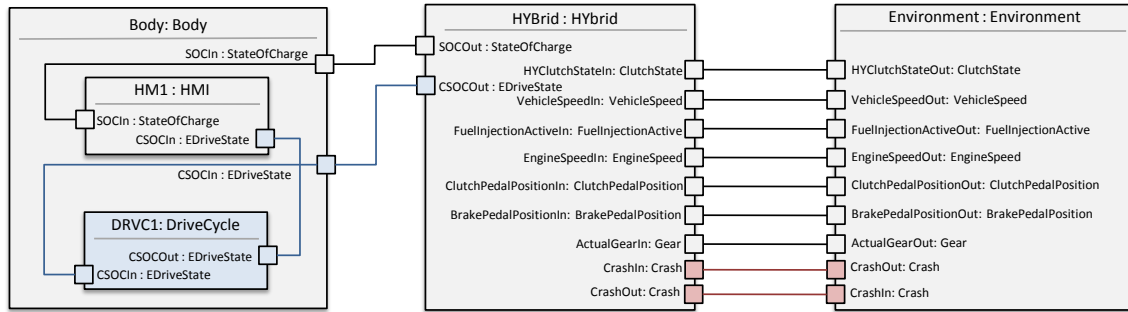
Package name	Implemented [%]	[# metaclasses]	Note
Function modeling (FDA)	73,33	15	-
Feature modeling	50,00	8	-
Vehicle feature modeling	33,33	3	-
Variability	69,23	13	-
Summary [%]	64,10		

Table A.7.: Implementation status for EAST-ADL part of the mapping process

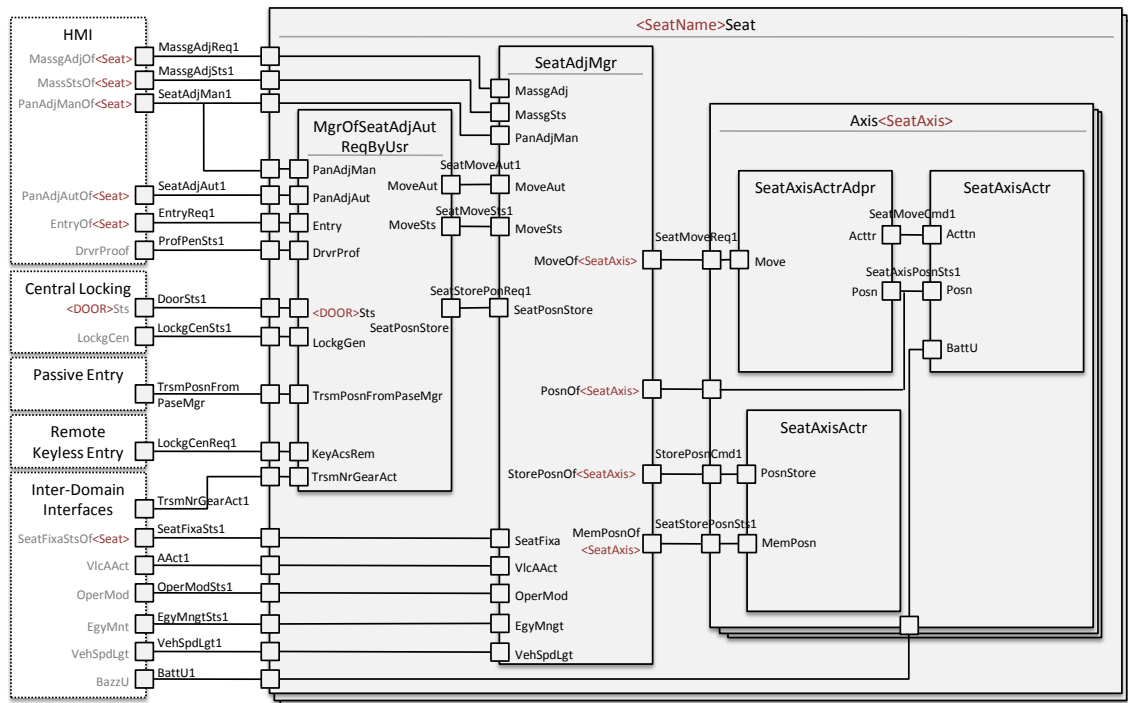
Note that attributes of meta classes are not considered in the calculation.

A. Appendix

A.4. Evaluation Use Cases

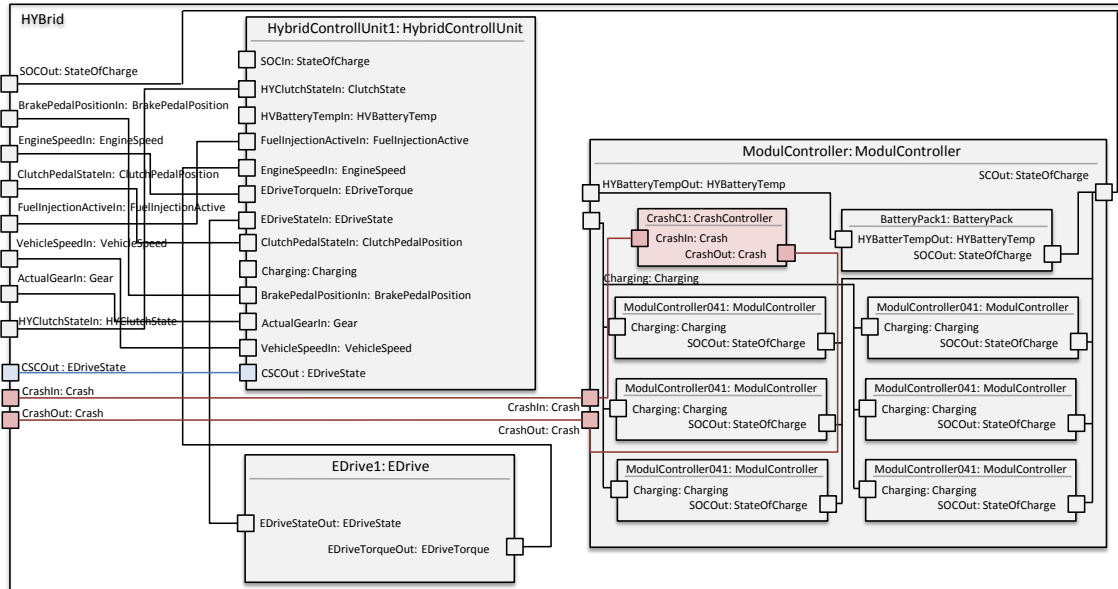


(a) Evaluation use case 1: *HYBRID* subsystem

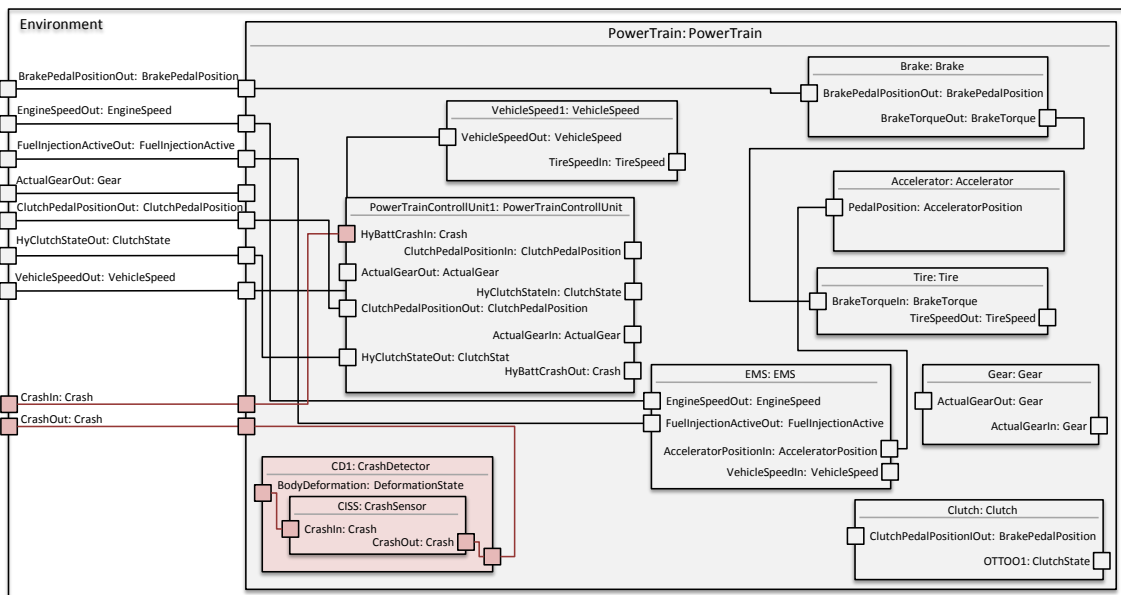


(b) Evaluation use case 2: Seat adjustment subsystem, [AUT09c]

Figure A.1.: Evaluation use cases



(a) Evaluation use case 1: *HYBRID* subsystem



(b) Evaluation use case 1: *Environment* subsystem

Figure A.2.: Evaluation use case 1: subsystems (soft real-time (blue) and hard real-time requirements from Section 5.2.2)

Bibliography

- [AQST10] A. Albinet, L. Quran, B. Sanchez, and Y. Tanguy. Requirement Management from System Modeling to AUTOSAR SW Components. *Embedded Real Time Software and Systems*, 2010.
- [AUT08] AUTOSAR. Technical overview. <http://www.autosar.org/>, 2008. Release 3.0, Document version 2.2.1.
- [AUT09a] AUTOSAR. *AUTOSAR Generic Structure Template*, 2009.
- [AUT09b] AUTOSAR. *AUTOSAR Software Component Template*, 2009.
- [AUT09c] AUTOSAR. *Explanation of Application Interfaces of the Body and Confort Domain*, 2009.
- [AUT09d] AUTOSAR. *Virtual Function Bus*, 2009.
- [Bar09] Thomas Barthel. Fibex - Datamodel for ECU Network Systems. www.asam.net, 2009.
- [BB01] Felix Bachmann and Len Bass. Managing variability in software architectures. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, SSR '01, pages 126–132, New York, NY, USA, 2001. ACM.
- [BCD⁺00] Len Bass, Paul Clements, Patrick Donohoe, John McGregor, and Linda Northrop. Fourth Product Line Practice Workshop Report. *Software Engineering Institute*, February 2000.
- [BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI series in software engineering. Addison-Wesley, 2003.
- [BK04] Kerstin Buhr and Ramin Tavakoli Kolagari. Softwarebasierte Produktlinien - Szenarien für Automobilhersteller und Zulieferer. Technical report, Gesellschaft für Informatik e.V. (GI), Bonn, 2004.
- [CFJ⁺08] P. Cuenot, P. Frey, R. Johansson, H. Lönn, M-O Reiser, D. Servata, R. Tavakoli Koligari, and D.J. Chen. Developing Automotive Products Using the EAST-ADL2, and Autosar Compliant Architecture Description Language. *European Congress on Embedded Real-Time Software (ERTS). Toulouse, France*, 2008.
- [CFJ⁺10] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lönn, Yianis Papadopoulos, Mark-Oliver Reiser, Anders Sandberg, David Servat,

Bibliography

- Ramin Tavakoli Kolagari, Martin Törngren, and Matthias Weber. The EAST-ADL Architecture Description Language for Automotive Embedded Software. In *Proceedings of the 2007 International Dagstuhl Conference on Model-based Engineering of Embedded Real-Time Systems*, MBEERTS'07, pages 297–307, Berlin, Heidelberg, 2010. Springer-Verlag.
- [CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.
- [CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45:621–645, July 2006.
- [Con10a] The ATESSST2 Consortium. *EAST-ADL Domain Model Specification, Deliverable D4.1.1*, Juni 2010. ATESSST Deliverable D4.2.1 V1.0.
- [Con10b] The ATESSST2 Consortium. *Evaluation Report of EAST-ADL2 Variability and Reuse Support, Deliverable D2.1*, May 2010. ATESSST Deliverable D2.1.
- [Cvl] CVL Homepage. User Guide. www.variabilitymodeling.org/, visited in July 2010.
- [Cza04] Krzysztof Czarnecki. Overview of generative software development. In *In Proceedings of Unconventional Programming Paradigms (UPP) 2004, 15-17 September, Mont Saint-Michel, France, Revised Papers*, pages 313–328. Springer-Verlag, 2004.
- [DB07] Mark Dalgarno and Dr. Danilo Beuche. Variant management. *British Computer Society*, 2007.
- [DDN07] Paul Grünbacher Deepak Dhungana, Rick Rabiser and Thomas Neumayer. Integrated Tool Support for Software Product Line Engineering. *ASE*, pages 533–534, 2007.
- [ea08] Hand Blom et al. *Reuse Guide. ATESSST Deliverable D4.2.2*, January 2008.
- [Eas] Enterprise Architect Homepage. Feature List. <http://www.sparxsystems.com/>, visited in July 2010.
- [EV05] Jacky Estublier and German Vega. Reuse and variability in large software applications. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 316–325, New York, NY, USA, 2005. ACM.
- [FBC06] Davide Falessi, Martin Becker, and Giovanni Cantone. Design Decision Rationale: Experiences and Steps Ahead Towards Systematic Use. *SIGSOFT Softw. Eng. Notes*, 31, September 2006.

- [FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, Software Engineering Institute, 2006.
- [FLV06] Peter H. Feiler, Bruce A. Lewis, and Steve Vestal. The SAE architecture analysis design language (AADL) a standard for engineering performance critical systems. In *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pages 1206–1211. IEEE, October 2006.
- [GG06] C. J. Michael Geisterfer and Sudipto Ghosh. Software component specification: A study in perspective of component selection and reuse. In *Proceedings of the Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*, pages 100–, Washington, DC, USA, 2006. IEEE Computer Society.
- [Has10] Andreas Haselsberger. Design and Implementation of a Domain Specific Architecture for Programmable Logic Controllers. Master’s thesis, Institute for Technical Informatics, Graz University of Technology, 2010.
- [Hau06] M. Hause. The SysML modelling language. In *5th European Systems Engineering Conference*, September 2006.
- [HKM06] William A. Hetrick, Charles W. Krueger, and Joseph G. Moore. Incremental return on incremental investment: Engenio’s transition to software product line practice. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, OOPSLA ’06*, pages 798–804, New York, NY, USA, 2006. ACM.
- [HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE’08)*, pages 943–944, New York, NY, USA, May 2008. ACM.
- [Hon09] Uwe Honekamp. The Autosar XML Schema and Its Relevance for Autosar Tools. *IEEE Softw.*, 26:73–76, July 2009.
- [Kae09] Gerald Kaeding. Produktlinien im Automobilbereich. Technical report, Universität Koblenz-Landau, 2009.
- [KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering, ICSE ’08*, pages 311–320, New York, NY, USA, 2008. ACM.
- [KCH⁺90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA). Technical report, Software Engineering Institute, 1990.

Bibliography

- [KF09] Olaf Kindel and Mario Friedrich. *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. dpunkt, Heidelberg, 2009.
- [Kol06] Ramin Tavakoli Kolagari. *Requirements Engineering für Software-Produktlinien Eingebetteter, Technischer Systeme*. Fraunhofer IRB Verlag, 2006. Fraunhofer IESE, Kaiserslautern; Univ. of Kaiserslautern, Computer Science Department, AG Software Engineering.
- [KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, March 2008.
- [Lei09] Andrea Leitner. A Software Product Line for a Business Process Oriented IT Landscape. Master's thesis, Institute for Technical Informatics, Graz University of Technology, 2009.
- [LGLH08] Wenjing Li, Yucheng Guo, Weizhi Liao, and Rongwei Hang. Research on ontology component description model based on the semantic web. In *Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference*, pages 697–702, Washington, DC, USA, 2008. IEEE Computer Society.
- [MA09] Cem Mengi and Ibrahim Arma. Ein Klassifikationsansatz zur Variabilitätsmodellierung in E/E-Entwicklungsprozessen. In *Software Engineering*, pages 125–130, 2009.
- [Mag] MagicDraw Homepage. Feature List. <http://www.magicdraw.com/>, visited in July 2010.
- [Mat04] Mari Matinlassi. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. In *Proceedings of the 26th International Conference on Software Engineering, ICSE '04*, pages 127–136, Washington, DC, USA, 2004. IEEE Computer Society.
- [Met] MetaCase Homepage. MetaEdit+® Workbench and Modeler. <http://www.metacase.com/>, visited in July 2010.
- [MORST09] Matthias Biehl Mark-Oliver Reiser, Helko Glathe, David Servat, and Yann Tanguy. *The EAST-ADL Analysis Platform*, November 2009. ATESSST Deliverable D4.3.1 V1.0.
- [MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26:70–93, January 2000.
- [Mul10] Gerrit Muller. Light weight architecture: The way of the future?, 2010.
- [Nor08] Linda Northrop. Software product lines essentials. Technical report, Carnegie Mellon University, 2008.

- [NXX10] Xu Nianfang, Yang Xiaohui, and Li Xinke. Software components description based on ontology. In *Proceedings of the 2010 Second International Conference on Computer Modeling and Simulation - Volume 04*, ICCMS '10, pages 423–426, Washington, DC, USA, 2010. IEEE Computer Society.
- [Oaw] oAW Homepage. Feature List. <http://www.openarchitectureware.org/>, visited in July 2010.
- [OB02] Rob C. van Ommering and Jan Bosch. Widening the scope of software product lines - from variation to composition. In *Proceedings of the Second International Conference on Software Product Lines*, SPLC 2, pages 328–347, London, UK, UK, 2002. Springer-Verlag.
- [ODF07] Camille Salinesi Olfa Djebbi and Gauthier Fanmuy. Industry Survey of Product Lines Management Tools: Requirements, Qualities and Open Issues. *15th IEEE International Requirements Engineering Conference*, pages 301–306, 2007.
- [OMG07] OMG. *OMG Unified Modeling Language (OMG UML)*, 2007. Infrastructure, V2.1.2.
- [OMG10] OMG. *OMG Systems Modeling Language*, 2010. Specification Version 1.2.
- [OT10] F. Ougier and F. Terrier. Eclipse Based Architecture of the EDONA Platform for Automotive System Development. *Embedded Real Time Software and Systems*, 2010.
- [Pap] Papyrus Homepage. Papyrus for EAST-ADL. <http://www.papyrusuml.org/>, visited in July 2010.
- [PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
- [Pre10] Christopher Preschern. Piscas - pisciculture automation system. Master's thesis, Institute for Technical Informatics, Graz University of Technology, 2010.
- [Pura] pure::systems Homepage. Product Demonstration. <http://www.pure-systems.com/fileadmin/downloads/pv-presentation-de.pdf>, visited in July 2010.
- [Purb] pure::systems Homepage. Video Material. <http://www.pure-systems.com/>, visited in July 2010.
- [RKW09] Mark-Oliver Reiser, Ramin Tavakoli Kolagari, and Mathias Webber. Compositional variability - concepts and patterns. In *Proceedings of the 42nd Hawaii International Conference on System Sciences*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

Bibliography

- [SAB09] Adel Smeda, Adel Alti, and Abdellah Boukerram. An environment for describing software systems. *W. Trans. on Comp.*, 8:1610–1619, September 2009.
- [Sit] *Software Product Lines. Technical Management*. http://www.sei.cmu.edu/productlines/frame_report/tool_support.htm, visited in July 2010.
- [SN10] Carl-Johan Sjöstedt and Tahir Naseer. *The EAST-ADL Modeling Workbench.*, May 2010. ATESSST Deliverable D4.2.1 V1.0.
- [SZ10] Jörg Schäuffele and Thomas Zurawka. *Automotive Software Engineering*. Vieweg + Teubner, Wiesbaden, 4. edition, 2010.
- [tre09] EB tresos. Autosar at a glance. <http://www.eb-tresos-blog.com/technologies/autosar/>, December 2009. visited 25.08.2010.
- [vdBBFR03] Michael von der Beeck, Peter Braun, Ulrich Freund, and Martin Rappl. Architecture Centric Modeling of Automotive Control Software. In *SAE Technical Paper Series 2003-01-0856*, 2003.
- [vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35:26–36, June 2000.
- [WL99] D. M. Weiss and R. Lai, editors. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.
- [WR02] B. Whittle and M. Ratcliffe. Software component interface description for reuse. In *Software Engineering Journal*, pages 307 – 318. IEEE Computer Society, 2002.
- [ZS07] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik: Protokolle und Standards*. Vieweg, Wiesbaden, 2. edition, 2007.