(ITEA 2 – 13017)

Enabling of Results from AMALTHEA and others
for Transfer into Application and
building Community around

Deliverable: D 1.1

Analysis of Necessary Design Steps

Work Package: 1

Continuous Design Flow and Methodology

Task: 1.1/1.2

Analyzing Concepts in the Design Flow & Identifying Design Steps
Analyzing Exchange of Development Artefacts between Organizations & Tools

Document Type:	Deliverable	**Classification:**	Public
Document Version:	Final	**Contract Start Date:**	01.09.2014
Document Preparation Date:	30.09.2015	**Duration:**	31.08.2017

History

Rev.	Content	Resp. Partner	Date
0.1	Set up Document	Jan Jatzkowski	April 10, 2015
0.2	Updated document structure	Jan-Philipp Steghöfer	June 26, 2015
0.3	Added content about ISO 26262	Jan-Philipp Steghöfer	June 29, 2015
0.4	Added introduction, summary, and content about V-Model	Jan-Philipp Steghöfer	July 21, 2015
0.5	Added content about AUTOSAR	Philipp Heisig	July 22, 2015
0.6	Added first version of design steps	Salome Maro	July 29, 2015
0.7	Added content about SPES/SPES XT	David Schmelter	July 30, 2015
0.8	Refined design steps	Salome Maro	August 8, 2015
0.9	Example for exchange of development artefacts using AUTOSAR	Jan Jatzkowski	August 14, 2015
0.10	Refined design steps	Salome Maro	August 21, 2015
0.11	Added diagrams to illustrate use of design concepts in design steps	Jan-Philipp Steghöfer	August 26, 2015
0.12	Added description of product line engineering	Christopher Brink	August 31, 2015
0.13	Added introduction and description	Jan-Philipp Steghöfer	September 1, 2015
0.14	Added preliminary analysis of ISO 26262 compatibility	Salome Maro, Maria Trei	September 14, 2015
0.15	Finishing touches	Jan-Philipp Steghöfer	September 19, 2015

Contents

List of Figures

List of Tables

Summary

This document describes a number of design steps that are common in the development of embedded automotive systems. They are presented together with the goals that drive them, the artefacts used and provided, as well as the tools used in them. In context of AMALTHEA4public, these design steps establish which parts of the design flow must be supported by the AMALTHEA platform as well as allow the identification of needs for traceability.

In addition to the design steps, this document also includes requirements and use cases for the exchange of artefacts between design steps and potentially between different companies involved in the development process. Furthermore, a preliminary analysis of the conformance of the identified design steps with the ISO 26262 standard has been conducted.

1 Introduction

The purpose of this document is to provide an overview of the design steps that are common in the engineering of embedded automotive systems. These design steps are usually part of a development process, but the lifecycle aspect is not regarded here. Instead, we focus on the individual steps, the artefacts they use and provide, the goals they help achieve and the tools that are used.

The overview will be beneficial to the project in several aspects:

1. The existing AMALTHEA tool chain can be evaluated against the designs steps and checked for their coverage.
2. The artefacts exchanged between design steps and thus potentially between different companies involved in a joint development effort are identified and characterised.
3. The requirements for traceability between artefacts created in the different design steps can be expressed in terms of their lifecycle, their producers and consumers.
4. The requirements for traceability between various tools can be established.

As such, the overview of design steps will be an important input to other work packages, including WP5 in which the tool chain is developed. Within WP1, the document will be used as the foundation for the definition of a traceability concept (D1.2) as well as for an integrated design flow (D1.3).

1.1 Methodology

This document is the result of an extensive data collection process in which the project partners in AMALTHEA4public provided detailed information about the design steps they follow when developing multi-core embedded software, especially in the automotive domain. The following project partners contributed to the data collection effort:

- Robert Bosch GmbH
- Timing Architects
- rt-labs
- IFAK
- University of Gothenburg
- Dortmund University of Applied Sciences and Art
- University of Paderborn

- AVL Turkey
- Fraunhofer IPT, Project Group "Mechatronic Systems Design"
- BHTC

While the academic partners provided information mostly from their experience with different industrial partners they cooperate with in research projects, the industrial partners contributed with their concrete practical experience. In addition, the project partners OFFIS and TWT contributed a preliminary analysis of the identified design steps w.r.t. their compatibility with ISO 26262.

The data collection and analysis process is illustrated in Figure 1.1. Data was collected using forms that had specific fields for the different aspects of the steps such as stakeholders, goals, tools, etc. We asked all project partners to fill out these forms. They were accompanied by an informative guideline which explained the purpose of the data collection and what is expected from the partners. We prototyped both form and guideline with our project partners from rt-labs AB and refined the documents according to their feedback. The refined form and guideline was sent by email to all the partners. Phone calls were scheduled with all participating project partners that allowed us to present the purpose of the study and answer all queries on how to fill the forms. The partners had one month to fill out the forms and send them back, again via email.

The collected data was analysed by first reading through all the documents and collecting questions on matters that were unclear or needed more explanation. These questions were sent back to the respective partners for clarification. After feedback from the partners, all unique steps that appeared for one or more project partners where included in the analysis. This analysis identified design steps, design concepts, and tools which are common among the partners. These were critically studied to understand if they where similar enough to be grouped and represented by one step without distorting the meaning given by the partners. The main criteria for grouping steps was the similarity of the description and/or goal.

The analysis produced the set of the design steps that are presented in this document. It includes both steps that are common in all the companies, such as requirements engineering, and also those that are specific to companies adopting a certain paradigm of software development. For instance, deriving product variants is a step specific to companies which use product lines. Design concepts have been identified and grouped as well, along with the tools that are used to produce and consume them. An important subgroup of design concepts are (design) *artefacts*. Artefacts are created when the design steps are followed. In contrast, there are other concepts that are merely used as input, such as guidelines and templates.

We furthermore analysed how artefacts are exchanged between the design steps by using the information from the data collection to derive use cases and requirements for exchange. The information was also used to compile an overview of the artefact exchange between design steps. The Software Process Engineering Meta-Model (SPEM, cf. [20]) was used for the purpose of representing the elements of the design steps in diagrams. The design steps where modelled as *Activities* that group other elements such as other activities. Thus, an activity is a building block of a process. Artefacts that are used in the design steps without being created by any such step (e.g., templates for requirements or reviews) are modelled as *Guidance*. Artefacts that are the result of a design step are modelled as *Work Products*. The *Tools* used in the design steps are also included. This kind of modelling allows for using the created method content for process tailoring purposes and to combine it with a model of a lifecycle easily.

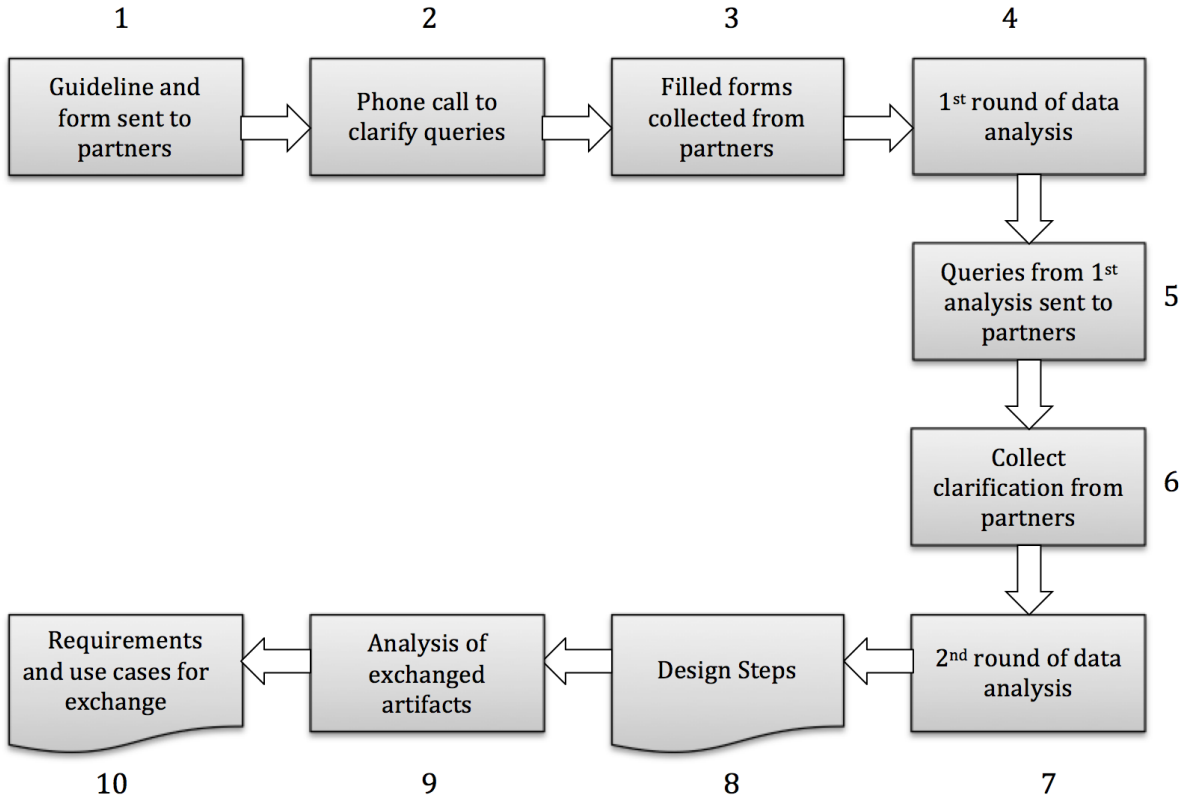

Figure 1.1: Data collection and analysis process.

1.2 Document Structure

The rest of this document is structured as follows: Chapter 2 gives an overview of common development processes and engineering standards in the automotive domain, thus providing the context in which the design steps are applied. The design concepts, design goals, and the design steps are introduced in Chapter 3. We report on requirements and use cases for the exchange of design artefacts in Chapter 4 and present a preliminary analysis of the compatibility of the identified design steps with ISO 26262 in Chapter 5. The report concludes with a discussion and an outlook on further work in AMALTHEA4public.

2 Development Methodologies and Standards

While the design steps are currently regarded isolated from the lifecycle they are embedded in, the project partners use them within the context of a development methodology or development standard. This section gives an overview of the methodologies and standards currently in use within the AMALTHEA4public consortium. They define the context in which the design steps must be seen and will — at the same time — form the foundation of the design flow at the end of the project in D1.3.

In this context, ISO 26262, a standard for safety in road vehicles, plays an especially prominent role since the AMALTHEA platform will be made compatible with it. Current work in Work Package 4 "Safety" has analysed the gaps between AMALTHEA and the standard and effort will be spent in closing these gaps, both from a technological as well as from a methodological perspective. Therefore, a preliminary analysis of the compliance of the design steps is included in this report in Chapter 5. The standard is introduced here briefly in Section 2.4

2.1 V-Model

The V-Model can be either seen as an abstract definition of the dependencies in a software or system development lifecycle or as a concrete development methodology. In the former understanding, analysis, design, and implementation activities on the left branch of the "V" are juxtaposed with testing and validation activities on the right side of the "V" as depicted in Figure 2.1. In the latter understanding, this abstract view is augmented with very concrete design steps. Examples for such methodologies are the V-Modell XT [12], prescribed for the development of complex technical systems by the German government and the V-Model (see, e.g., [16]) used by the federal US government for the same purpose. These specific versions of the V-Model are of limited interest for the purpose of this report and we focus on the more abstract understanding.

In the automotive domain, many development efforts follow the abstract V-Model. In fact, both the AUTOSAR methodology and the ISO 26262 standard are based on the V-Model. The dependencies between hardware and software development make it necessary to reduce the development risks early on, create detailed specification, and follow a somewhat rigid process. On the other hand, the safety requirements make it necessary to have detailed tests and validation activities that allow verification and validation of the specifications and the code at all stages of the product lifecycle. The "V" visualises how these activities must be coordinated and which stages of the validation correspond to which stages of the specification.

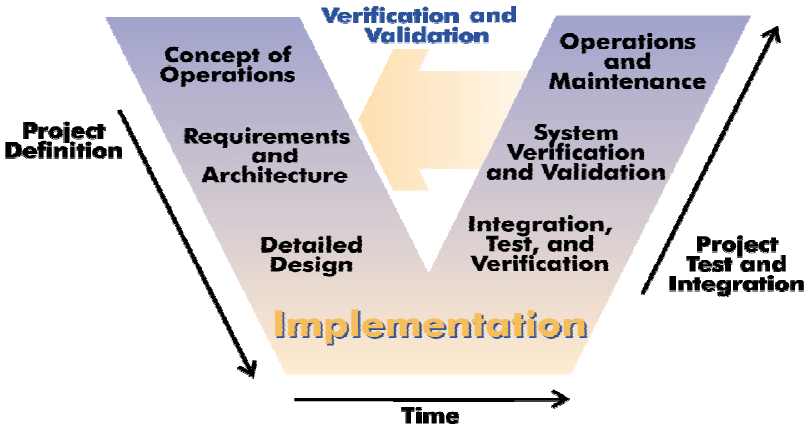

Figure 2.1: The abstract view of the V-Model, showing the two juxtaposed branches [21].

2.2 SPES / SPES XT

SPES and SPES XT are the results of the research project SPES 2020[1] which was sponsored by the German Federal Ministry of Education and Research. The SPES Modeling Framework provides a method for the development of software-intensive embedded systems. It has been developed by 21 partners from academia and industry from November 2009 until January 2012 [23]. The following elucidations are based on [22, part II].

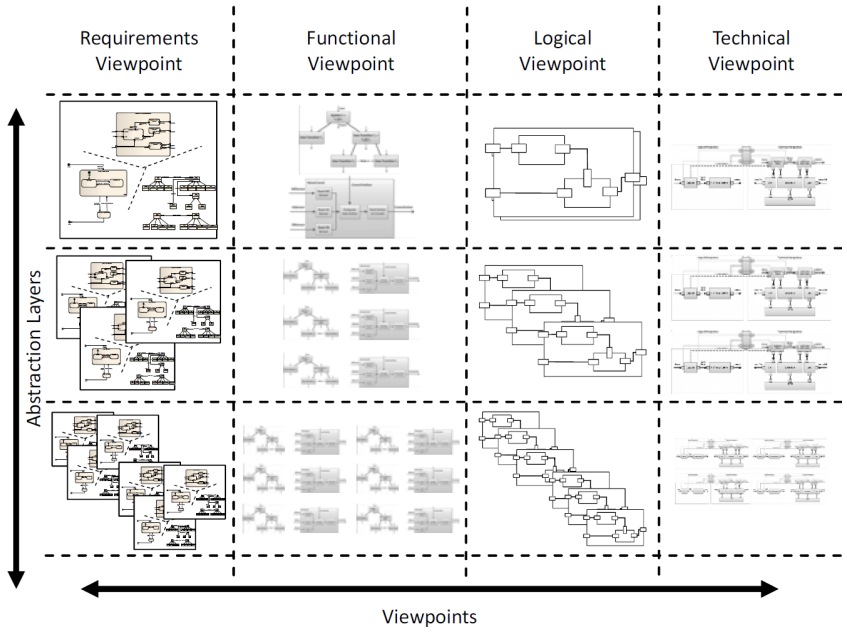

Figure 2.2: The SPES Modeling Framework, from [22, sec. 3.5]

SPES aims to provide a development method that is independent from the application domain: To this end, it defines the two fundamental concepts *Viewpoints* and *Abstraction Layers* which

[1] Software Plattform Embedded Systems 2020

form a two dimensional design space (cf. Figure 2.2). SPES promotes a seamless model-based development approach facilitating reuse and automation. In context of the automotive domain SPES aims to be AUTOSAR compliant.

Viewpoint: A viewpoint in SPES follows the notion of the IEEE 1471 standard "*Recommended Practice for Architecture Description of Software-Intensive Systems*" [1] and can be understood as a template or pattern for the development of individual views on the system under development and its environment. SPES focuses on the following four viewpoints: *Requirements Viewpoint*, *Functional Viewpoint*, *Logical Viewpoint* and *Technical Viewpoint*. These are detailed further in the following paragraphs.

Abstraction Layer: Each system or system element can be modelled on different levels of abstraction. Increasing the level of detail (and at the same time decreasing the abstraction layer) adds knowledge to the according design element. Abstraction Layers in SPES are user defined, i.e. application domain specific. For example, Abstraction Layers in the automotive domain might be "Supersystem", "System", "Subsystem", and "Hardware/-Software Component". Mappings between the different abstraction layers allow tracing of the appropriate refinements.

The following paragraphs illustrate the four Viewpoints SPES focuses on in more detail.

Requirements Viewpoint: The requirements viewpoint defines concepts and techniques for systematically eliciting and specifying the requirements of the system (-element) under development. It defines four models: The *Context Model* captures information and constraints about the systems environment. SPES utilizes different languages for capturing context information. For instance, SysML BDDs can be used to model static context information, Petri nets or finite state machines can be used to capture the dynamic aspects. The *Goal Model* describes the intentions of the different stakeholders. KAOS goal diagrams, i* models or SysML requirement diagrams can be used to create Goal Models in SPES. *Scenario Models* specify exemplary interactions of the system with its environment. SysML Sequence Diagrams or ITU Message Sequence Charts can be used to create this type of requirements artefact. Finally, the *Solution-oriented Requirements Model* represents a first step towards the later systems implementation. It consists of a structural, operational and behavioural part utilizing SysML BDDs, Activity Diagrams and State Machines as modelling languages.

Functional Viewpoint: The Functional Viewpoint provides a formal and model-based behaviour specification for the system under development. It provides two model types that structure the behavioural requirements according to user functions and provide an abstract realization of these. First, the *Functional Black Box Model* is created. It formalizes the requirement models as those user functions that can be observed at the systems boundary. Based on the black box model, the *Functional White Box Model* refines the user functions. The purpose of this model is to provide a decomposition of the user functions into smaller functional units in order to give an abstract description of the realization of the user functions.

Logical Viewpoint: The Logical Viewpoint describes the internal logical structure and the behaviour of the system under development. It is a platform independent model, abstracting

from a concrete hardware platform. The main model type of the logical viewpoint is the logical component architecture: it describes the logical components of the system realizing the functions from the Functional Viewpoint as well as their relationships. The logical components form an acyclic hierarchical tree with the leaves being atomic logical components that are not decomposed any further. Behaviour, e.g., in terms of state machines, is associated to atomic logical components only.

Technical Viewpoint: The technical viewpoint addresses the question on how to get from the platform-independent models (cf. Logical Viewpoint) to platform-specific models, i.e., it handles the actual deployment of logical components. To this end, it supports the modelling of Resources (like computational power, bandwidth, storage, ...), Schedulers (i.e., distribution of Resources) and Tasks. The technical Viewpoint can be compared to the AUTOSAR Virtual Function Bus (cf. section 2.3).

In addition to the aforementioned Viewpoints and Abstraction Layers, SPES also addresses cross-cutting concerns like safety and real-time.

SPES XT is the follow-up project of SPES. It also was sponsored by the German Federal Ministry of Education and Research and has been executed by 21 partners from academia and industry from May 2012 until July 2015 [23]. SPES XT is structured in three dimensions: *Engineering Challenges* (modular proof of safety, optimal deployment, mechatronics and software, variant management, interconnection, early validation), *Cross-Cutting Issues* (methodology for holistic model-based development, tools and tool-platforms, systematic transfer into practice) and *Application Domains* (use cases "Automotive System Cluster" and "Desalination Plant") [24].

2.3 Autosar

AUTOSAR is the de-facto standard within the automotive industry for developing electrics/-electronics architectures by describing software, hardware and operating system parts. The demand for such a specification is motivated by an increasing functional complexity within automotives. Therefore, the main objective of AUTOSAR is to manage complexity in a cost efficient way by, among others, standardizing the component-based software architecture of ECUs to support collaboration and exchange between various partners including OEMs and suppliers, reusing artefacts (e.g., software components or hardware description) within different vehicles and platform variants, defining an open architecture, and providing basic system functions in a standardized way.

The main feature of AUTOSAR is the layered ECU software architecture which allows a hardware-independent development of component-based vehicle applications. To achieve this, a software abstraction layer, called Runtime Environment (RTE), abstracts the hardware-oriented Basic Software (BSW) from the hardware-independent software layer. The BSW provides all system services and functions for communication, memory management, diagnostic and so forth. Further, it contains several layers for the abstraction of the underlying ECU hardware by separating the ECU layout definition (connections between periphery and micro-controller) from upper layers as well as providing specific hardware drivers. In order to enable the description of functional interaction between software application components independently from the ECU allocation, AUTOSAR introduced the Virtual Function Bus (VFB) [9]. This concept is

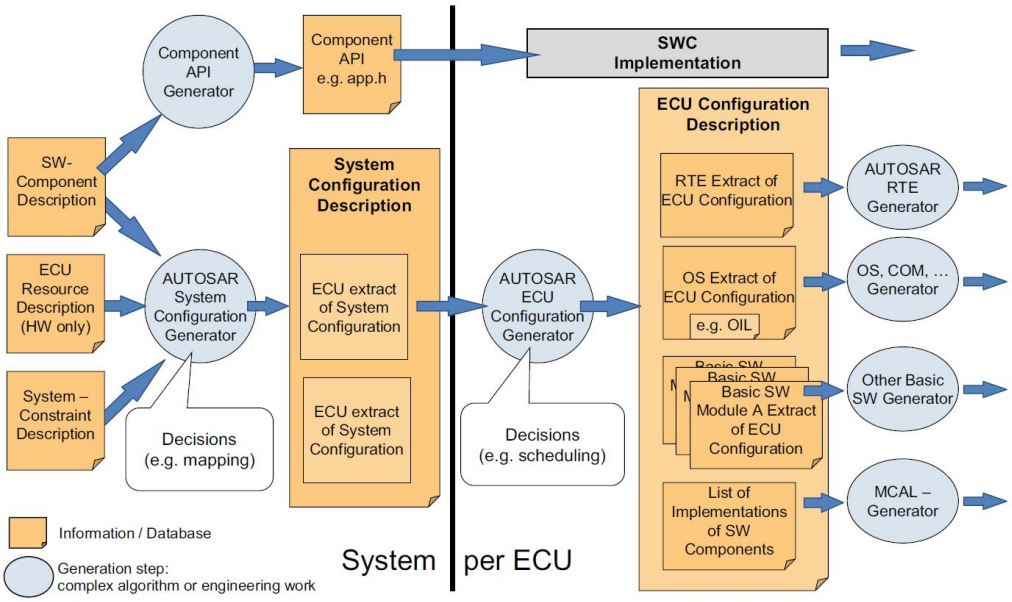

Figure 2.3: The AUTOSAR methodology for developing automotive software [7]

implemented by the RTE and realizes the handling of vehicle-wide functions by providing capabilities for the communication between the software application components as well as granting access to the BSW services.

Apart from the software architecture, AUTOSAR also standardises the development methodology for automotive software [8], which includes coverage of special aspects in order to allow the implementation of components into a ECU as well as the integration of ECUs in the vehicles communication network of different bus systems. Furthermore, support for activities and their dependencies among other, descriptions and usage of tools is given. As shown in Figure 2.3, several development domains like the definition of the VFB or the ECU configuration are covered within the methodology. The information flow within the methodology is standardised by means of a XML-based data exchange format, while a meta model ensures a formal description.

The initial starting point of the methodology is the definition of input descriptions: While the *SW-Component Description* provides a standardized component model with well-defined interfaces and hardware requirements, the *ECU Resource Description* specify the available hardware together with their characteristics in a hierarchical manner (e.g., micro-controller with two cores and flash memory of 512 KB). In addition, information regarding the network topology (interconnections between ECUs), communication matrix, used protocols (e.g., CAN, FlexRay), attributes (e.g., data rates, timing), and so on are given within the *System Constraint Description*. These descriptions function as input for the generation of the system configuration, where software component descriptions are distributed to the different ECUs and ports are mapped to communication signals. For this step, defined constraints as well as ECU resources have to be taken into account. Based on the system configuration, the BSW and RTE of each ECU are then configured. Finally, the components of the software application have to be implemented in order to allow the generation of software executables. Since the release of specification 4.0, AUTOSAR also supports the definition of product lines through an integrated variability management [6], which can optionally be managed by feature models [5].

2.4 ISO 26262

The international standard ISO 26262 is a safety standard with the general title "Road vehicles – Functional safety". It was published in 2011, based on the functional safety standard IEC 61508, which deals with safety of E/E systems in every area of industry. In contrast to this, IS0 26262 concentrates on the development of electrical and/or electronic systems of passenger cars with a maximum gross vehicle mass up to 3500kg. It is expected that future versions or extensions of the standard will have a greater coverage and also include vehicles like trucks and motorcycles.

ISO 26262 describes the safety lifecycle of automotive E/E systems, including management, development, production, operation, service and decommissioning. A central element is the hazard analysis and risk assessment in the beginning of the safety lifecycle, where so-called Automotive Safety Integrity Levels (ASILs) are defined. Based on this classification, requirements for avoidance of unreasonable residual risk are defined and validation methods are recommended or prescribed.

In addition, intended relations with suppliers and other stakeholders are described, which helps to supports integration of an item at product and vehicle level. To this end, the concept of *Safety Elements out of Context* (SEooC) is defined in the standard. Figure 2.4 provides an overview of the different parts of ISO 26262.

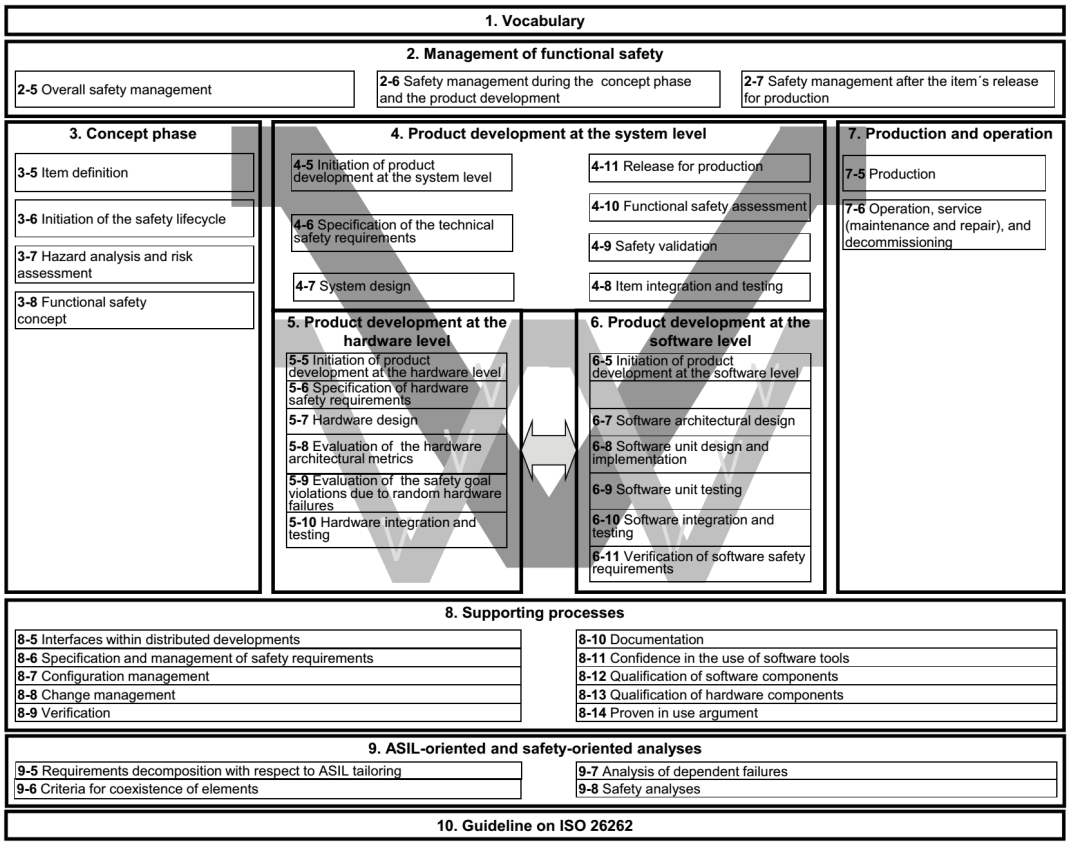

Figure 2.4: Overview of the different parts of ISO 26262 [17]

2.5 Product Line Engineering Process

Product lines were introduced for the development of complex and variable software systems [14]. Within these product lines common and variable parts of a system are described for example in the form of feature models [18]. These models consist of a hierarchically arranged set of features connected through different types of associations. Feature models can be used both in development as well as in the later product configuration [2, 13]. In order to allow the derivation of configured products, features are connected to software development artefacts. Afterwards, a product of the product line can be generated from any valid combination of features. Due to the nature of a product line, the whole process consists of two fundamental steps, the development process (Domain engineering) and the configuration process (Application engineering) [10]. While the development process includes steps to identify requirements and to develop the product line platform including the required features, the configuration process includes the product generation based on a customer's needs (Figure 2.5).

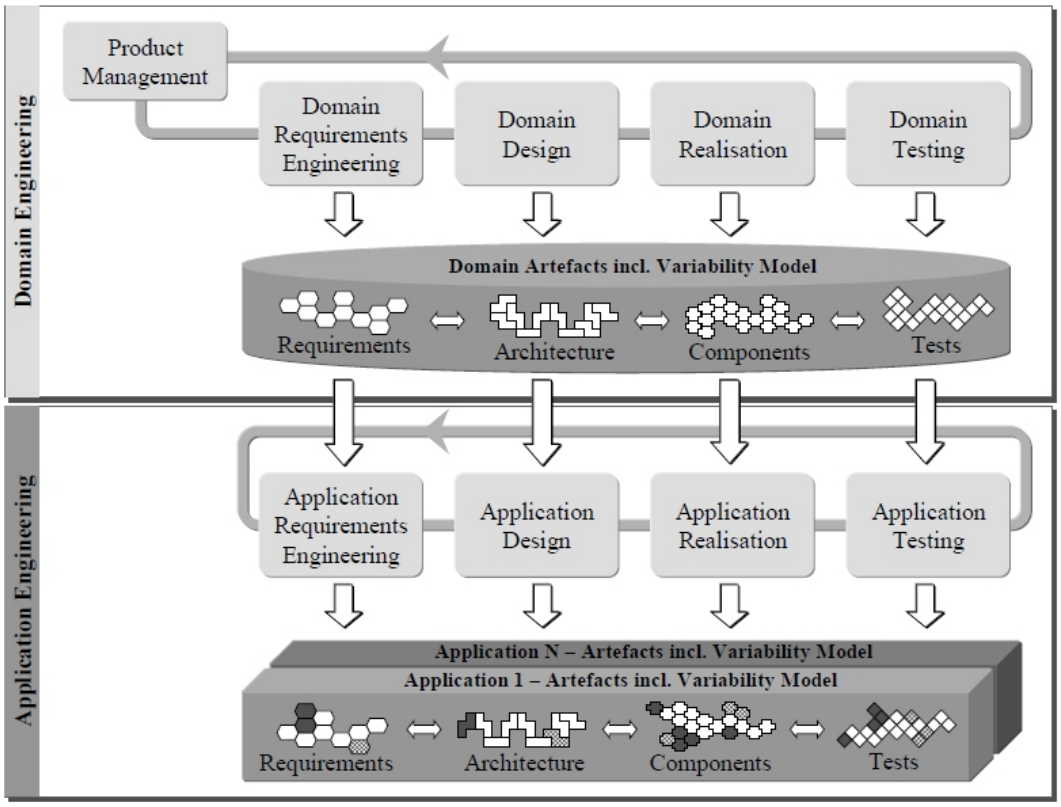

Figure 2.5: Software product line development process [10]

Domain Engineering In the domain engineering process a reusable platform (reference architecture) with common and variable parts is developed. In order to develop this platform, the standard software development steps requirements engineering, design, realization and testing are performed. In addition, a step for managing the product portfolio of the company is added. The main goals of this process are the definition of the product line scope as well as modelling of common and variable parts. Further, the development and test of reusable artefacts is focused.

Application Engineering The application engineering allows to build a product out of the

product line by using the reference architecture. Therefore, the customer's requirements on the product are determined in a first step. Based on the requirements, necessary parts of the reference architecture as well as necessary software components will be selected and configured. Furthermore, non-existing product-specific components are developed during the application realization phase. The process ends with the testing of the complete application, including its architecture and components.

3 Design Steps, Goals and Concepts

When developing software every step that is taken is done to achieve a specific goal or purpose. This chapter describes the design steps and sub-activities that are carried out to accomplish design goals when developing embedded software. The concepts which are used and provided by each step and sub-activity are also described in detail. A description of the tools that are used to create and facilitate use of these design concepts is also given.

This section uses SPEM[1] notation to illustrate the design steps. An overview of the notation can be found in Table 3.

3.1 Overview of the design steps

Figure 3.1 gives an overview of the design steps that are the results of the analysis of the data collected from the project partners. Note that in this deliverable we do not imply the order in which these steps are carried out since defining a concrete development process with a concrete lifecycle is generally company specific. While some steps are carried out sequentially, others can be done in parallel. The dependencies between the design steps at times also imply an iterative approach where the design steps are repeated or at least revisited after other work has been performed. Such an approach is very common in iterative-incremental lifecycles. Having said that, the concrete life cycle in which the design steps are placed is not regarded here and a wide variety of lifecycles can be applied. For an overview, please refer to Chapter 2.

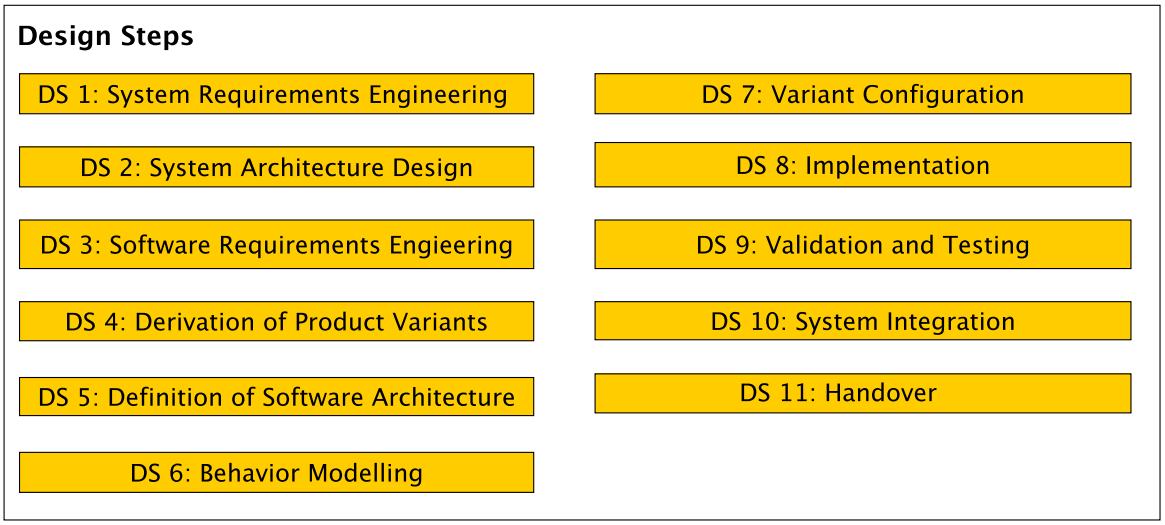

Figure 3.1: Overview of the identified Design Steps

[1]Software & Systems Process Engineering Metamodel (`http://www.omg.org/spec/SPEM/2.0/`)

Concept	Description	Icon
Task	Describes a unit of work. A task can create or transform a work product and can be assigned to a certain role. It can contain individual steps that have to be performed as part of carrying out the task. Guidances can be provided, e.g., to give guidelines or checklists. Tasks can be subsumed in activities.	
Activity	Groups other elements such as method content uses, milestones, or other activities. Thus, an activity is both a building block of a process or a process in itself. The elements within an activity can be linked to role uses, work product uses, and process parameters. The order of elements within the activity is denoted as the *work breakdown structure* and is established by defining predecessors for the elements in the structure. The "Use"-elements below are always defined within the context of an activity.	
Process	Defines the structure of activities and tasks that determine the order in which sequences of work are performed and phases and milestones are completed to get to the final product. Within a process, concrete role uses, task uses, work products, etc. are defined. A customised process for a specific project is modelled as a *Delivery Process*.	
Work Product	Documents, models, code, or other tangible elements that are produced, consumed, and modified by tasks. Responsibilities for work products can be assigned to roles.	
Role	Denotes an individual or a group of individuals with a certain set of skills, competencies, and responsibilities required in the process. Different roles can be filled by different people during the process and an individual can fill several roles if required.	
Guidance	Provides additional information about the elements in the method content. Different kinds of guidances are possible: guidelines, templates, checklists, tool mentors, supporting materials, reports, concepts, examples, and others.	

Table 3.1: SPEM concepts used in the description of the design steps

The identified steps cover most aspects of a traditional software development effort, starting from contract negotiation and scope identification and ending at the delivery of the software. Software maintenance has not been covered in the data collection. While many of the steps are generic in the sense that they could occur very similarly in any domain for which software is developed, there are some more specific steps and circumstances that can be explained in the context of the automotive domain.

First of all, the design steps reflect the differentiation of system and software. The steps DS-1 and DS-2 refer to requirements engineering and architecture for the entire system, including the software but mainly focused on the hardware. Our analysis shows that software requirements are elicited in a separate step, DS-3. Later on, artefacts pertaining to the system (e.g., the DS1 System Model) are used to validate the software and to integrate the software with the hardware in DS10.

Another re-occurring theme is product line issues and variants. There are distinct steps for the identification and refinement of variants (DS4 and DS7) as well as specific steps or consideration in the architectural decisions (DS5.3) and the design of the software's behaviour (DS6.1). The artefact DC13 Variant Model is thus also one of the most re-used artefacts and exchanged between a number of design steps.

Finally, since data from companies that work in a customer-centric way (e.g., rt-labs AB) and with in-house product development (e.g., Bosch) have been combined, it can be seen that the roles of "Customer" and "Technical Expert" as well as "Product Manager" are all used. Depending on whether the product under development is for an external customer or the requirements are set according to an internal product development and diversification strategy, the stakeholders differ. Even for customer-driven companies, internal product line strategies might exist and corresponding stakeholders must be consulted. An overview of the different roles identified in the analysis and their involvement in the different design steps is given in Figure 3.2 and Figure 3.3.

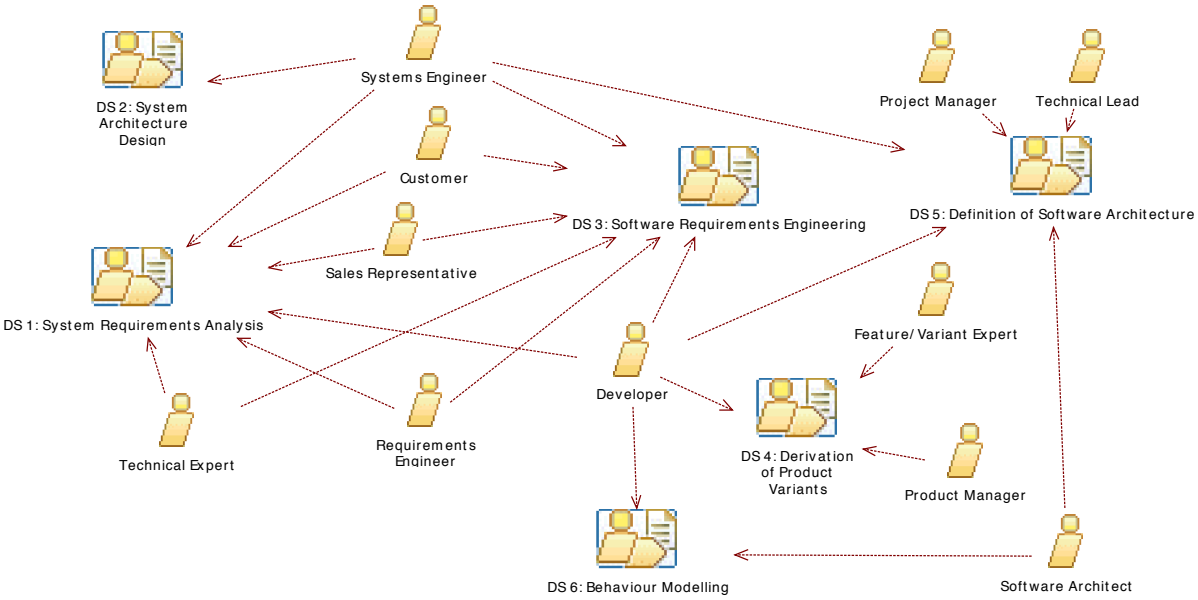

Figure 3.2: Different roles and their involvement in design steps DS1 to DS6.

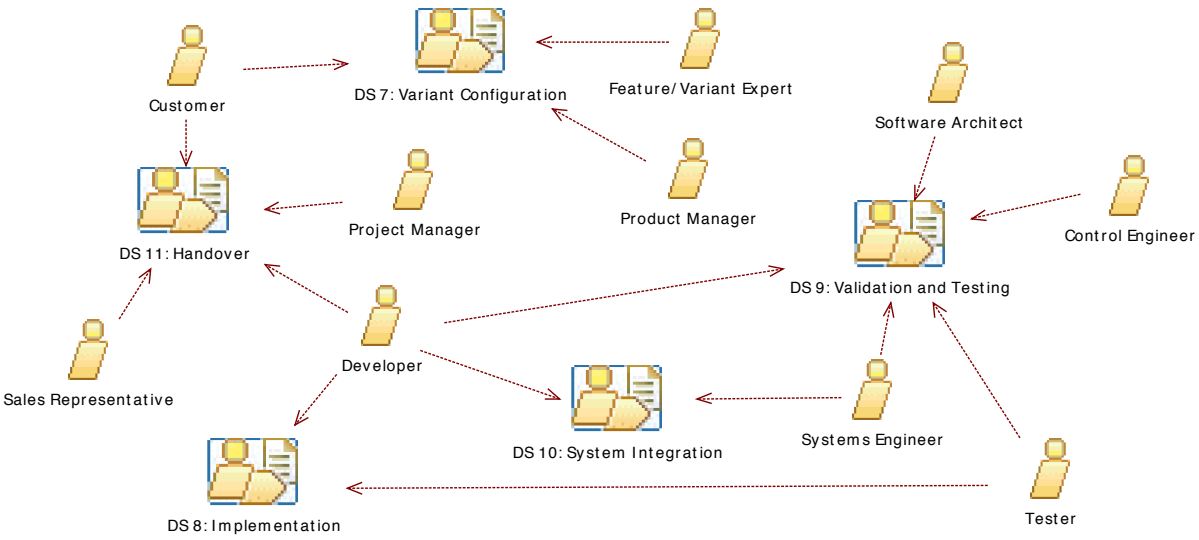

Figure 3.3: Different roles and their involvement in design steps DS7 to DS11.

The design steps have been classified to show which aspects they address. These aspects reflect broad topical areas that are usually covered by people working in roles specific to them. The following classes have been identified:

Systems Engineering is concerned with aspects of the overall system, including its hardware part.

Product Line Engineering addresses issues of variants and features.

Business aspects are concerned with the negotiation of contracts, sales, and customer relations.

Software Engineering relates to all issues pertaining to the design, construction, and validation of the developed software.

3.2 Design Step 1: System Requirements Analysis

ID: DS 1	Name: System Requirements Analysis	Involved Roles: Systems Engineers, Sales Representatives, Technical Experts, Developers, Customers, Requirements Engineers
Tag	Systems Engineering, Product Line Engineering	
Goal	To capture requirements of the System Under Development (SUD)	
Description	When designing a system for the automotive domain, several disciplines such as electronics, software engineering and control engineering are involved. In this step, overall requirements for the system from the various domains are elicited. Here, the requirements can be described textually as well as model-based. The aim is to come up with a model and/or text document describing the entire system under development. This can then be used as starting point for eliciting specific requirements of all the various disciplines involved.	
Sub-Activities	DS 1.1 System Requirements Elicitation DS 1.2 Definition of Platform Requirements DS 1.3 Definition of Product Requirements	
Uses	DC 1: Customer Requirements DC 2: Requirements Collection Template	
Provides	DC 3: System Requirements Model DC 4: System Requirements Specification Document	

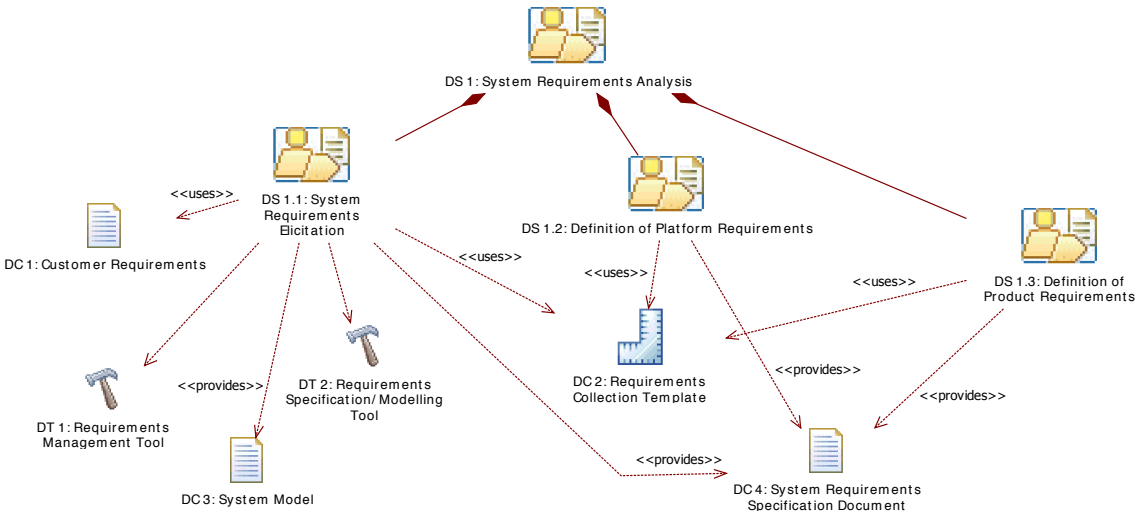

Figure 3.4: The steps, concepts, and tools used in design step DS1.

3.2.1 Detailed Steps

ID: DS 1.1	Name: System Requirements Elicitation	Involved Roles: Systems Engineers, Sales Representatives, Technical Experts, Developers, Customers, Requirements Engineers
Tag	Systems Engineering	
Description	The first step in systems engineering is the system requirements elicitation. The system requirements can be captured in forms of text or models depending on the requirements engineering method chosen by the responsible roles. System requirements describe the entire system as a black-box, focusing mainly on what the system does (goals and scenarios), who are the users and what other systems will it interact with [11]. Model Based Systems Engineering (MBSE) methods provide a set of partial models to capture system requirements. Examples of the partial models are Environment models which describe the system under development (SUD) in its context, Application scenarios which capture the different use cases of the SUD and requirements models which capture additional functional and non-functional requirements.	
Uses	DC 1: Customer Requirements DC 2: Requirements Collection Template	
Tools	DT 1: Requirements Management Tools DT 2: Requirements Specification/Modelling Tools	
Provides	DC 3: System Requirements Model DC 4: System Requirements Specification Document	

ID: DS 1.2	Name: Definition of Platform Requirements	Involved Roles: Sales Representatives, Technical Experts, Developers
Tag	Product Line Engineering	
Description	This step captures the requirements relating to the product line. Platform requirements describe requirements that refer to the platform product line. This platform is common to all products and corresponds to a reference architecture.	
Uses	DC 2: Requirements Collection Template	
Provides	DC 4: System Requirements Specification Document	

ID: DS 1.3	Name: Definition of Product Requirements	Involved Roles: Sales Representatives, Technical Experts, Developers
Tag	Product Line Engineering	
Description	In this step the requirements based on the differences between the platform/product line and the individual product are defined. Product requirements are requirements that apply only to a variant that is not included in all products of the product line. However, it is possible that this variant is used in more than one product. Thus, product requirements are requirements for an extension of the platform.	
Uses	DC 2: Requirements Template	
Tools	DT 1: Requirements Management Tools DT 2: Requirements Specification/Modelling Tools	
Provides	DC 4: System Requirements Specification Document	

3.2.2 Design Concepts

ID: DC 1	Name: Customer Requirements
Description	These are the requirements from a customer describing what the desired system should do.

ID: DC 2	Name: Requirements Collection Template
Description	This template provides a format the requirements structure should follow. It can be created using various requirements management tools.

ID: DC 3	Name: System Requirements Model
Description	The system requirements model provides a common understanding of the SUD and serves as a starting point for the discipline-specific development. It comprises several partial models, e.g., for modelling the environment of the SUD, requirements, application scenarios etc.

ID: DC 4	Name: System Requirements Specification Document
Description	This artefact contains the elicited system requirements. In many cases it is generated from a requirements management tool.

3.2.3 Design Tools

ID: DT 1	Name: Requirements Management Tools
Description	This a tool that is used to enter and manage requirements. It can be a standalone requirements management tool or part of an application lifecycle management tool. Some examples of these tools are IBM Jazz DOORS NG (`https://jazz.net/`), ProR (`http://www.eclipse.org/rmf/pror/`), Trac (`http://trac.edgewall.org`) and PTC Integrity (`http://www.ptc.com/application-lifecycle-management/integrity`).

ID: DT 2	Name: Requirements Specification/Modelling tools
Description	These are tools that allow users to create and edit requirements models such as use case models and sequence diagrams. Examples of these tools are Papyrus(`https://eclipse.org/papyrus/`) and Scenario Tools (`http://scenariotools.org`)

3.3 Design Step 2: System Architecture Design

ID: DS 2	Name: System Architecture Design	Involved Roles: Systems Engineers
Tag	Systems Engineering	
Goal	To develop an overall system architecture	
Description	In this step an overall system architecture is designed according to the system requirements elicited. This system architecture consists of several partial models that describe the system, sub-systems and the relations and interactions of the system to the environment. Both the structure and behaviour of the SUD are modelled in in this step. Models such as the active structure model or SysML blocks are used to model the structural view of the SUD, while models such as activity diagrams and state machines are used to model behaviour.	
Uses	DC 3: System Requirements Model DC 4: System Requirements Specification Document	
Provides	DC 5: System Architecture	

3.3.1 Design Concepts

ID: DC 5	Name: System Architecture
Description	This is the architecture of the entire SUD. It consist of various models describing the structure and behaviour of the SUD.

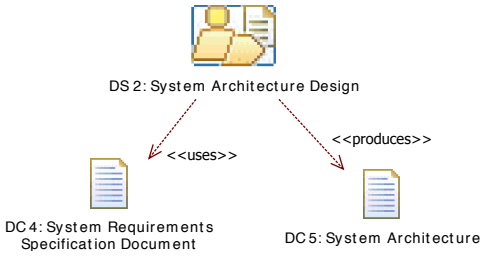

Figure 3.5: The steps, concepts, and tools used in design step DS2.

3.4 Design Step 3: Software Requirements Engineering

ID: DS 3	Name: Software Requirements Engineering	Involved Roles: Sales Representatives, Technical Experts, Developers, Systems Engineers, Requirements Engineers, Customers
Tag	Software Engineering, Business	
Goals	• To properly understand and evaluate the required systems software (in terms of features and cost) before committing to developing it. • To develop a refined and formal software requirements specification	
Description	The responsible roles work out and analyse the necessary *software* requirements based on the system requirements document (if available). Requirements are also collected from customers or potential users through several iterations of meetings. The collected requirements are recorded in a Software Requirements Specification Document. They might also be captured in a formal representation, e.g. in terms of Modal Sequence Diagrams. Acceptance tests for the software to be developed are defined. A quotation is prepared using the requirements, acceptance tests and estimated effort. The customer signs an agreement indicating the approval of the quotation.	
Sub-Activities	DS 3.1: Software Requirements Elicitation DS 3.2: Preparation of Requirements Specification Document DS 3.3: Definition of Acceptance Tests DS 3.4: Preparation of Quotation	
Uses	DC 2: Requirements Collection Template DC 3: System Requirements Model DC 4: System Requirements Specification Document DC 5: System Architecture DC 11: Task Tickets DC 12: Milestones	
Tools	DT 1: Requirement Management Tools DT 2: Requirements Specification/Modelling tools	
Provides	DC 6: Software Requirements Specification Document DC 7: MSD Specification DC 8: Acceptance Tests DC 9: Quotation DC 10: Purchase Order (Agreement)	

3.4.1 Detailed Steps

ID: DS 3.1	Name: Software Requirements Elicitation	Involved Roles: Sales Representatives, Technical Experts, Developers, Systems Engineers, Requirements Engineers
Classification	Software Engineering	
Description	The responsible person arranges meetings with customers. In each of the meeting he/she tries to understand the customer needs by interviewing the customer, showing various demos, and collecting any artefacts that may be useful. In cases where the system requirements already exist from the system requirements elicitation step, software requirements can be elicited from the system requirements.	
Uses	DC 2: Requirements Collection Template DC 3: System Requirements Model DC 4: System Requirements Specification Document DC 5: System Architecture	
Provides	DC 6: Software Requirements Specification Document (Early version)	

ID: DS 3.2	Name: Preparation of Requirements Specification Document	Involved Roles: Sales Representatives, Technical Experts, Developers, Systems Engineers, Requirements Engineer
Classification	Software Engineering	
Description	The software requirements which where collected from the customer and also those derived from system requirements are recorded in a requirements management system. The requirements are usually in formal requirements specification formats such as use cases or user stories. For companies using model-based engineering approaches and developing technical systems that require real time coordination these requirements can, e.g., be specified in terms of a Modal Sequence Diagram (MSD) specification. MSDs are a mechanism for formally specifying requirements on the interaction of components in a system. Constraints such as timing and safety are also input into the requirements management tool as non functional requirements. If a requirement is linked to other artefacts the traces to the artefacts are also created in this step. When all the requirements have been recorded, the software requirements specification document can be created by exporting the requirements in formats like PDF, Word, ReqIF etc. The cost of the software and the time estimate for development of the software is calculated and also included in the document.	
Uses	DC 2: Requirements Collection Template	
Tools	DT 1: Requirements Management Tools DT 2: Requirements Specification/Modelling tools	
Provides	DC 6: Software Requirements Specification Document DC 7: MSD Specification	

ID: DS 3.3	Name: Definition of Acceptance Tests	Involved Roles: Sales Representatives, Technical Experts, Developers, Systems Engineers, Requirements Engineers
Classification	Software Engineering	
Description	From the requirements, acceptance tests are written that must pass when the system is completed. These tests also have to be agreed on by the customer so that when all tests pass, the system is complete and can be handed over to the customer. Depending on the system to be developed, the acceptance tests can be written as unit-tests, integration tests, functional tests, or a combination of these.	
Uses	DC 6: Software Requirements Specification Document	
Provides	DC 8: Acceptance Tests	

ID: DS 3.4	Name: Preparation of Quotation	Involved Roles: Sales Representatives
Classification	Software Engineering	
Description	The Sales Representative prepares a quotation based on the requirements, acceptance tests, estimated effort and delivery schedule. The customer agrees to the quotation by signing a purchase order.	
Uses	DC 6: Software Requirements Specification Document DC 8: Acceptance Tests DC 11: Task Tickets DC 12: Milestones	
Provides	DC 9: Quotation DC 10 : Purchase Order	

3.4.2 Design Concepts

ID: DC 6	Name: Software Requirements Specification Document
Description	This document contains a detailed description of all the requirements for the system to be built. The requirements are stored in a requirements management system. This document can be exported in various forms and given to the customer. It is used as a guidance of what the system should contain. The document also includes a budget and the time it will take for the system to be built.

ID: DC 7	Name: MSD Specification
Description	The MSD Specification comprises a set of formal requirements and is dedicated to the message-based coordination of different systems (or system parts). The MSD approach considers assumptions on the environment as well as real-time requirements and is applicable to hierarchical component architectures, which makes it well suited in the context of technical systems.

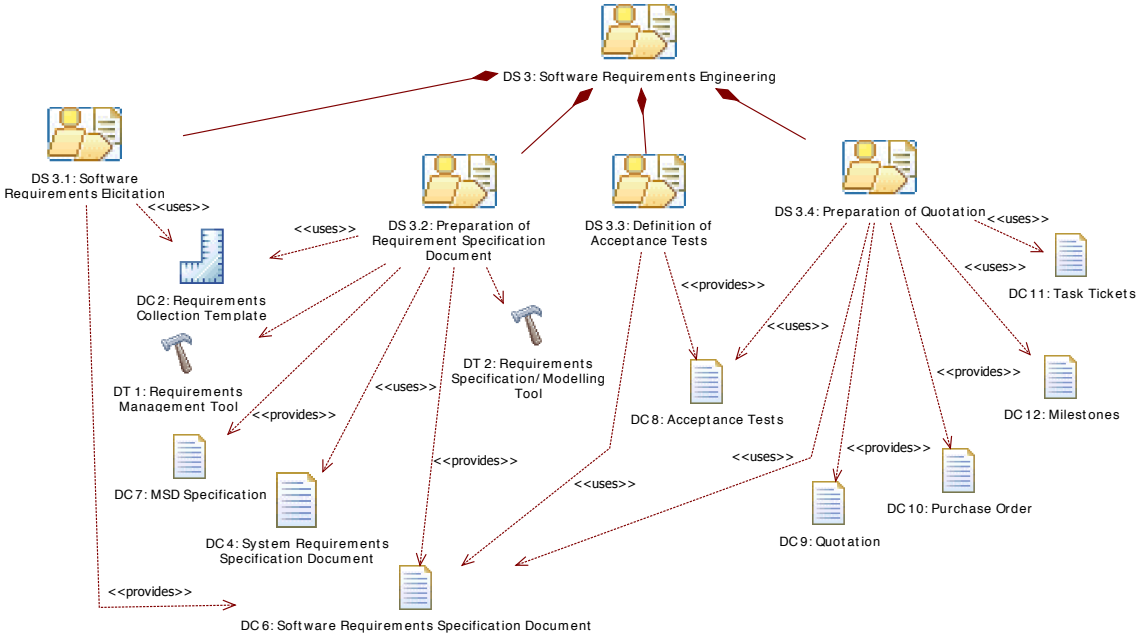

Figure 3.6: The steps, concepts, and tools used in design step DS3.

ID: DC 8	Name: Acceptance Tests
Description	Acceptance tests verify that the software satisfies the requirements agreed upon. These tests are run before the software is delivered to a customer. When all tests pass then the software satisfies all the requirements and can be handed over to the customer. The tests are also run internally during development to continuously validate the software being built. These tests can be implemented using appropriate test frameworks.

ID: DC 9	Name: Quotation
Description	The quotation specifies the cost for building the system, as defined by the requirements specification and acceptance tests. A delivery schedule is included based upon the planned milestones. The cost is calculated based upon the estimates.

ID: DC 10	Name: Purchase Order
Description	A document signed by the customer agreeing to the terms in the quotation.

ID: DC 11	Name: Task Tickets
Description	A set of tickets in a collaboration tool (e.g., Trac, JIRA, or a physical backlog), representing implementation activities. Each ticket has an estimation of effort attached to it, e.g., in number of hours of remaining implementation effort. The number is continuously updated during implementation and is used in burn-down charts.

ID: DC 12	Name: Milestones
Description	A milestone consists of a number of tasks to be completed at a certain date. The product of each milestone is usually a deliverable to the customer.

3.5 Design Step 4: Derivation of Product Variants

ID: DS 4	Name: Derivation of Product Variants	Involved Roles: Product Managers, Feature/Variant Experts, Developers
Classification	Product Line Engineering	
Goal	To model and describe variants of a product line	
Description	Variants within product lines represent particular product features. Together with a common platform of all products (commonalities), the use of variants results in distinct products. In this step, variants are defined based on the collected requirements. Afterwards, these variants are summarized together with the commonalities in order to create a variant model.	
Sub-Activities	DS 4.1: Definition of Software Variants DS 4.2: Definition of Hardware Variants	
Uses	DC 4: System Requirements Specification Document DC 6: Software Requirements Specification Document DC 14: Hardware Model	
Tools	DT 3: Product Line Tools	
Provides	DC 13: Variant Model	

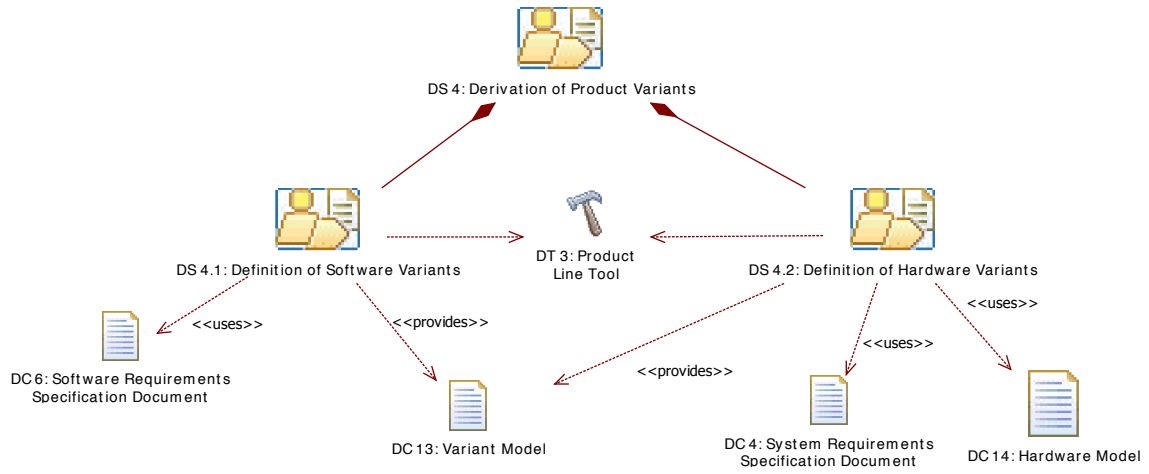

Figure 3.7: The steps, concepts, and tools used in design step DS4.

3.5.1 Detailed Steps

ID: DS 4.1	Name: Definition of Software Variants	Involved Roles: Product Managers, Feature/Variant Experts, Developers
Classification	Product Line Engineering	
Description	The software variants are described within a variant model which represents the relationships between the variants by its tree structure. In addition to the relationships, it is also possible to define dependencies between variants that are not in the same sub tree. These dependencies are defined by a constraint language. To determine them, it is necessary to analyse the product requirements for inconsistencies. Further, dependencies within the requirements have to be taken into account.	
Uses	DC 6: Software Requirements Specification Document	
Tools	DT 3: Product Line Tools	
Provides	DC 13: Variant Model	

ID: DS 4.2	Name: Definition of Hardware Variants	Involved Roles: Product Managers, Feature/Variant Experts, Developers
Classification	Product Line Engineering	
Description	In this step, hardware platforms and their variants are specified. This can be done in two different ways: On the one hand, a new hardware platform can be modelled. This platform is not associated with a specific hardware model. This initially abstract hardware model (which shows which hardware components are needed) can be linked with a concrete and matching hardware model that already exists later on. Nevertheless, the link is optional. On the other hand, a hardware platform can be generated by reading a (concrete) hardware model. The variant consists of all hardware elements and attributes of the hardware model. It is also possible to define additional hardware variants. These are used to specify which elements and associated attributes within a hardware platform can vary and if dependencies to other variants exist. The hardware variants are described by a model that allows a hierarchy of elements. In addition, the definition of varying elements or groups of elements is covered by the model. However, the description of hardware variants is not as powerful as the description for the software variants. This is due to the fact that hardware platforms usually contain only a small number of variants whereas software variants can occur in several ways. Furthermore, dependencies between elements are not specifiable in the model; this task has to be taken over by the linked hardware model. Only dependencies of hardware variants are covered by the hardware variability model.	
Uses	DC 4: System Requirements Specification Document DC 14: Hardware Model [opt]	
Tools	DT 3: Product Line Tools	
Provides	DC 13: Variant Model	

3.5.2 Design Concepts

ID: DC 13	Name: Variant Model
Description	A variant model covers all common and variable parts of a product line within one model.

ID: DC 14	Name: Hardware Model
Description	A model containing the description of the hardware components. It describes the structure of a hardware platform as well as its main attributes and characteristics, e.g., how many cores exist, how much memory is available, etc.

3.5.3 Design Tools

ID: DT 3	Name: Product Line Tools
Description	This is a tool that facilitates modelling of product lines. Such tools include features like variability modelling, product derivation from product lines, e.t.c. Examples of these tools are the AMALTHEA4public product line tool, Pure Variants (`http://www.pure-systems.com/pure_variants.49.0.html`) and BigLever (`http://www.biglever.com/overview/software_product_lines.html`).

3.6 Design Step 5: Definition of Software Architecture

ID: DS 5	Name: Define Software Architecture	Involved Roles: Software Architects, Developers, Systems Engineers, Technical Leads, Project Managers
Classification	Software Engineering	
Goal	• To model and describe various software components of the system and how they fit together. • To break down the requirements to a level where the implementation effort can be estimated in terms of time required for implementation. • To define the architecture of the system to be developed	
Description	The software is designed to implement the requirements. Here the components and function groups of the software and their relationships based on the requirements are modelled. In cases where product lines apply, the software component architecture for the product line also known as common software platform is defined. Here, it is necessary to provide required interfaces for software extensions of all individual products. For this purpose, first the component interfaces and common components are specified before their connections are defined. To determine the component architecture, software architects use the requirements as well as the corresponding variant model. Necessary components for all variable parts are also defined in the architecture.	
Sub-Activities	DS 5.1: Specification of Software Architecture DS 5.2: Review of Software Architecture DS 5.3: Refine Software Variants	
Uses	DC 6: Software Requirements Specification Document DC 7: MSD Specification DC 13: Variant Model DC 15: Software Architecture Review Protocol	
Provides	DC 11: Task Tickets DC 12: Milestones DC 16: Software Component Model DC 17: Software Architecture Document DC 18: Software Architecture Review Protocol Document	

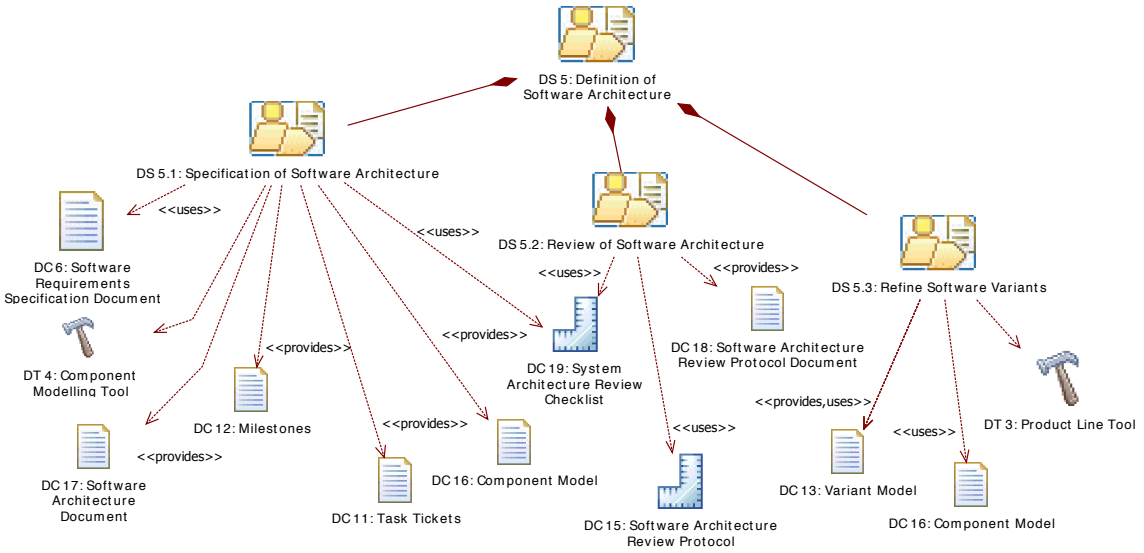

Figure 3.8: The steps, concepts, and tools used in design step DS5.

3.6.1 Detailed Steps

ID: DS 5.1	Name: Specification of Software Architecture	Involved Roles: Software Architects, Developers, Systems Engineers, Technical Leads, Project Managers
Classification	Software Engineering	
Description	From the specified requirements, the software architecture of the system is defined. The software architecture is usually defined using models such as component models which contains all the software components and their dependencies. Communication between these components is also specified in the architecture model. The architecture of a software can be described using more than one model in order to capture different perspectives of the software or to further refine it into a lower abstraction level. At this stage, tasks are created for how the requirements will be implemented with respect to the architecture. Each task results in a task-ticket. Milestones for the development project are also established in this step.	
Uses	DC 6: Software Requirements Specification Document DC 7: MSD Specification DC 19: System Architecture Review Checklist	
Tools	DT 4: Component Modelling Tools	
Provides	DC 11: Task Tickets DC 12: Milestones DC 16: Component Model DC 17: Software Architecture Document DC 46: Software Model	

ID: DS 5.2	Name: Review of Software Architecture	Involved Roles: Software Architects, Developers, Systems Engineers, Technical Leads, Project Managers
Classification	Software Engineering	
Description	The software architecture is reviewed by software architects to ensure that the architecture covers the requirements and that it is correct. If there are any mistakes the reviewers will mark them and/or suggest the changes to be made. The result of the review goes back to the people responsible for them to make the changes. This is therefore an iterative process.	
Uses	DC 6: Software Requirements Specification Document DC 7: MSD Specification DC 19: System Architecture Review Checklist DC 15: Software Architecture Review Protocol	
Provides	DC 18: Software Architecture Review Protocol Document	

ID: DS 5.3	Name: Refine Software Variants	Involved Roles: Software Architects, Developers, Systems Engineers, Technical Leads, Project Managers
Classification	Product Line Engineering	
Description	After all components have been defined, they are assigned to the corresponding software features in the variant model.	
Uses	DC 16: Component Model DC 13: Variant Model	
Tools	DT 3: Product Line Tools	
Provides	DC 13: Variant Model	

3.6.2 Design Concepts

ID: DC 15	Name: Software Architecture Review Protocol
Description	This is the document which describes the procedure for the review of software system architecture

ID: DC 16	Name: Component Model
Description	This is a model containing the software components of the system, their communication and interdependencies. It is usually exported from a specification/modelling tool such as Papyrus or Yakindu CoMo.

ID: DC 17	Name: Software Architecture Document
Description	This is the final document which describes the software architecture.

ID: DC 18	Name: Software Architecture Review Protocol Document
Description	This is the review protocol developed after the completion of software architecture review.

ID: DC 19	Name: System Architecture Review Checklist
Description	A reference list to check while determining the software architecture.

ID: DC 46	Name: Software Model
Description	Software Models provide an abstract description of the software, e.g., which runnables exist, how many instructions are required for the execution of each runnable, which date is communicated etc.

3.6.3 Design Tools

ID: DT 4	Name: Component Modelling Tools
Description	These are tools that are used to create component models of the software architecture. Examples for such tools are Papyrus or Yakindu CoMo.

3.7 Design Step 6: Behaviour Modelling

ID: DS 6	Name: Behaviour Modelling	Involved Roles: Developers, Software Architects
Classification	Software Engineering, Product Line Engineering	
Goal	To model and describe the functional behaviour of a system.	
Description	In this step the behaviour of the software components in the architecture is specified. Behaviour models describe the control structure of the system. In some cases the behaviour of non functional requirements is also specified.	
Sub-Activities	DS 6.1: Determine Component Behaviour DS 6.2: Algorithm Development DS 6.3: Determine Real Time Coordination Protocols DS 6.4: Identify Required Hardware	
Uses	DC 6: Software Requirements Specification Document DC 13: Variant Model DC 7: MSD Specification DC 3: System Model DC 20: Algorithm Development Review Checklist DC 21: Algorithm Development Review Protocol	
Provides	DC 22: Software Component Requirements Document DC 23: Algorithm Development Review DC 24: Component Tests DC 25: Behaviour Model DC 26: Real Time Coordination Protocols (as part of the software architecture)	

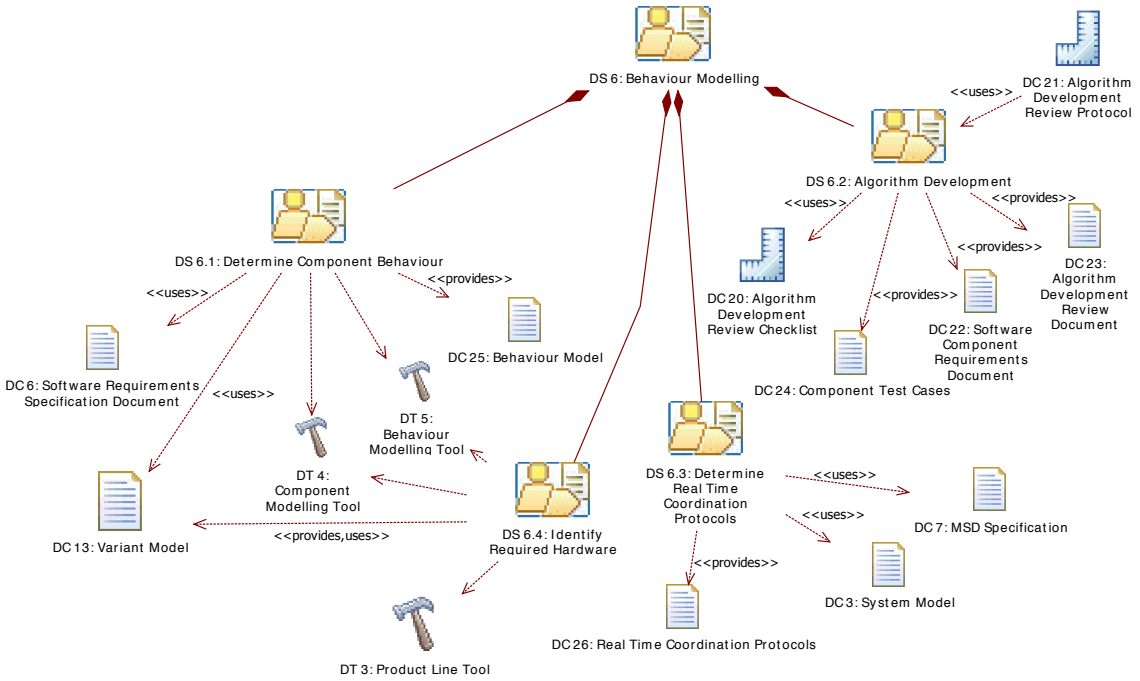

Figure 3.9: The steps, concepts, and tools used in design step DS6.

3.7.1 Detailed Steps

ID: DS 6.1	Name: Determine Component Behaviour	Involved Roles: Developers, Software Architects
Classification	Software Engineering	
Description	After the components architecture has been defined and associated with the variant model, now the behaviour of the variants is specified. Specifying the behaviour of components means specifying their control flow (when each step should be taken), data flow (steps that should be taken when certain input is produced) and state machines (steps that should be taken when certain events occur). Control flow is modelled using activity diagrams, data flow using flow charts or Petri nets and state machines using state charts.	
Uses	DC 6: Software Requirements Specification Document DC 7: MSD Specification DC 13: Variant Model	
Tools	DT 4: Component Modelling Tools DT 5: Behaviour Modelling Tools	
Provides	DC 25: Behaviour Model	

ID: DS 6.2	Name: Algorithm Development	Involved Roles: Developers, Software Architects
Classification	Software Engineering	
Description	Software developers and architects develop algorithms for desired software system requirements and create related component requirements.	
Uses	DC 20: Algorithm Development Review Checklist DC 21: Algorithm Development Review Protocol	
Provides	DC 22: Software Component Requirements Document DC 23: Algorithm Development Review Document DC 24: Component Tests	

ID: DS 6.3	Name: Determine Real Time Coordination Protocols	Involved Roles: Developers, Software Architects
Classification	Software Engineering	
Description	Based on the MSD specification, Real-Time Coordination Protocols are created to model reusable interaction and coordination behaviour between different software components. Real-Time Coordination Protocols are abstract protocols on application level that define the coordination between a pair of communicating roles and a connector that connects these roles. Communication is usually asynchronous and message-based, i.e., the sender can send its message independent of the receivers state, the sender of a message does not block while sending, and the receiver does not block while receiving.	
Uses	DC 3: System Model DC 7: MSD Specification	
Provides	DC 26: Real Time Coordination Protocols (as part of the software architecture)	

ID: DS 6.4	Name: Identify Required Hardware	Involved Roles: Software Architects, Developers, Systems Engineers, Technical Leads, Project Managers
Classification	Product Line Engineering, Systems Engineering	
Description	When modelling the behaviour, additional requirements relating to the hardware platform that must be met for this component may be identified. These requirements are specified by properties that are associated with the software variant in the variant model. The properties specify necessary hardware elements and attributes.	
Uses	DC 13: Variant Model DC 14: Hardware Model	
Tools	DT 3: Product Line Tools DT 4: Component Modelling Tools DT 5: Behaviour Modelling Tools	
Provides	DC 13: Variant Model	

3.7.2 Design Concepts

ID: DC 20	Name: Algorithm Development Review Checklist
Description	A reference list to check while developing the algorithms.

ID: DC 21	Name: Algorithm Development Review Protocol
Description	This is the document which describes the procedure of reviewing the developed algorithms.

ID: DC 22	Name: Software Component Requirements Document
Description	This is the final document which describes the requirements of each component in the software system.

ID: DC 23	Name: Algorithm Development Review Document
Description	This artefact describes the results of the algorithm development review.

ID: DC 24	Name: Component Tests
Description	These are the test cases which need to pass for component development to be accepted.

ID: DC 25	Name: Behaviour Model
Description	A model describing the functional behaviour of the system.

ID: DC 26	Name: Real Time Coordination Protocols (as part of the software architecture)
Description	This is the model of the software architecture containing the real time coordination protocols. It is usually in the form of sequence diagrams, interaction diagrams, and/or real time state charts.

3.7.3 Design Tools

ID: DT 5	Name: Behaviour Modelling Tools
Description	Tools for describing implementation aspects in a formal way. They can be used to model the expected behaviour of individual components and the entire system as well. Examples of these tools are YAKINDU Statecharts or Matlab Simulink.

3.8 Design Step 7: Variant Configuration

ID: DS 7	Name: Variant Configuration	Involved Roles: Product Managers, Feature/Variant Experts, Customers
Classification	Product Line Engineering	
Goal	Model and describe hardware platforms of the product line	
Description	In this step, a product from the product line is derived by selecting particular variants within the configuration process. Here, both the dependencies of software variants as well as the dependencies of hardware variations are taken into account. In addition, compatibility between software and hardware is considered by evaluating the properties. Therefore, dependencies of selected variants are immediately checked against the model. This includes dependencies which result from the tree structure as well as dependencies which have been defined by means of a constraint language. In case a hardware variant is selected, additional dependencies between these hardware versions are checked. In any case, the required properties of the software side are matched against the available properties on the hardware side and options that are not selectable are disabled. Thus, the selection of impossible combinations of variants is prevented, which can be modelled in two ways: on the one hand via restrictions of software variants and and on the other hand through restrictions of hardware variants. Following the configuration, a software component model for the selected variant is produced during the generation. It contains all the necessary components and their behaviour. This component model can be partitioned and distributed in the further development process. Furthermore, hardware platforms which matches the requirements are determined.	
Uses	DC 13: Variant Model	
Tools	DT 3: Product Line tools	
Provides	DC 14: Hardware Model DC 28: Software Component Model DC 27: Configured Variant Model	

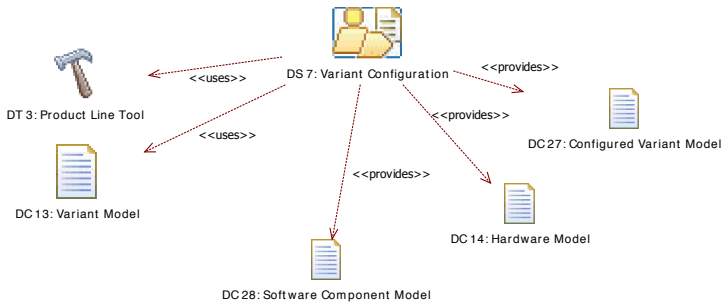

Figure 3.10: The steps, concepts, and tools used in design step DS7.

3.8.1 Design Concepts

ID: DC 27	Name: Configured Variant Model
Description	In this model, all variants have been resolved.

ID: DC 28	Name: Software Component Model
Description	This model is derived from the product line and contains only the software components which are necessary for a particular product.

3.9 Design Step 8: Implementation

ID: DS 8	Name: Implementation	Involved Roles: Developers, Testers
Classification	Software Engineering	
Goal	To implement the system.	
Description	In this step the software components are implemented. This can either be done by writing the code from scratch or generating the code via model-driven engineering approaches. This step is iterative and is repeated for each milestone.	
Sub-Activities	DS 8.1: Implementation of Software Components DS 8.2: Develop Tests DS 8.3: Software Integration DS 8.4: Code Review	
Uses	DC 11: Task tickets DC 12: Milestones DC 25: Behavioural Model DC 29: Basic Software Configuration DC 30: Defects DC 31: Implementation Review Checklist DC 32: Implementation Review Protocol	
Tools	DT 6: Collaboration Tools DT 7: Test Frameworks	
Provides	DC 30: Defects DC 33: Source Code DC 34: Unit Tests DC 35: Integration Tests DC 36: Integrated Software and Documentation	

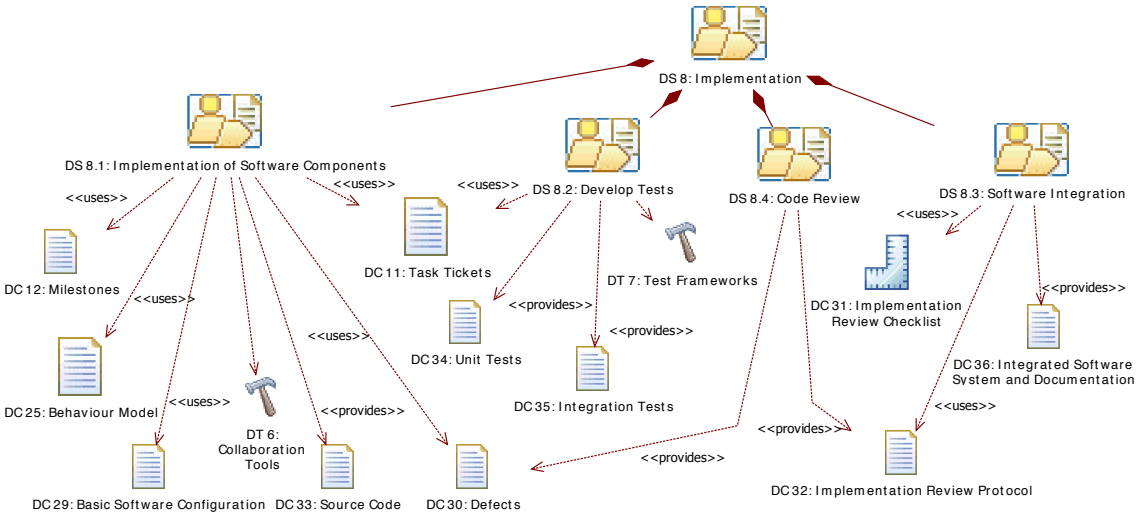

Figure 3.11: The steps, concepts, and tools used in design step DS8.

3.9.1 Detailed Steps

ID: DS 8.1	Name: Implementation of Software Components	Involved Roles: Developers
Classification	Software Engineering	
Description	The software components of the system are implemented by the developers. In traditional software development environments, developers write the code from scratch. In companies where model-driven software development is adopted, the code for the software components is generated from models using model transformation techniques.	
Uses	DC 11: Task Tickets DC 12: Milestones DC 25: Behavioural Model DC 29: Basic Software Configuration DC 30: Defects	
Tools	DT 6: Collaboration Tools	
Provides	DC 33: Source Code	

ID: DS 8.2	Name: Develop Tests	Involved Roles: Developers, Testers
Classification	Software Engineering	
Description	Software developers write unit tests as well as integration test for the software components that are being developed. Unit tests are used to test the functionality of individual components while integration tests are written to test the whole or parts of the system together. One or more tests should stress the system (e.g., with a high data rate) in order to reveal bugs that only surface in high-load situations.	
Uses	DC 11: Task Tickets	
Tools	DT 7: Test Frameworks	
Provides	DC 24: Component Tests DC 34: Unit Tests DC 35: Integration Tests	

ID: DS 8.3	Name: Software Integration	Involved Roles: Developers, Testers
Classification	Software Engineering	
Description	Software developers and testers integrate software components and make sure that the system works properly. This is done with the help of integration tests which can let the engineer know if the components work together or not. Integration is usually a continuous process and components are integrated as they come from the developers pipeline, potentially with the help of continuous integration tools such as Jenkins. The tools can also automatically execute integration tests.	
Uses	DC 31: Implementation Review Checklist DC 32: Implementation Review Protocol DC 35: Integration Tests	
Provides	DC 36: Integrated Software and Documentation	

ID: DS 8.4	Name: Code Review	Involved Roles: Developers, Testers
Classification	Software Engineering	
Description	The source code developed is peer reviewed and the results of the review are documented for further modification of the code.	
Uses	DC 33: Source Code	
Provides	DC 30: Defects DC 32: Implementation Review Protocol	

3.9.2 Design Concepts

ID: DC 29	Name: Basic Software Configuration
Description	This is a configured hardware-specific software layer. It describes software modules mapped to specific hardware.

ID: DC 30	Name: Defects
Description	Defects are problems found during or after implementation. Defects are entered into the issue-tracking system as tickets and are estimated and assigned to milestones.

ID: DC 31	Name: Implementation Review Checklist
Description	Before integrating the software system, the conditions which should be checked are mentioned in this list.

ID: DC 32	Name: Implementation Review Protocol
Description	This is the document which describes the procedure for the review of the software integration.

ID: DC 33	Name: Source Code
Description	Source code, e.g., source and header files for the programming language C (*.c, *.h)

ID: DC 34	Name: Unit Tests
Description	A unit test tests one source code module in isolation. Unit tests should be run automatically when sources or tests change and are checked in to the revision control system.

ID: DC 35	Name: Integration Tests
Description	Integrations tests test the whole system or several parts of the system together. Integration tests should also be automated as far as possible, however they often require manual interaction with hardware, which can make full automation difficult.

ID: DC 36	Name: Integrated Software System and Documentation
Description	This is the final SW system and its Documentation

3.9.3 Design Tools

ID: DT 6	Name: Collaboration Tools
Description	These are tools that facilitate storage and collaboration on project planning and development artefacts. The tools allow various users to work together on the project and provide functionalities like version control, continuous integration and automatic build of projects. Examples of these tools are Trac or JIRA for ticket management and connection to the version control software, Subversion or Git for versioning, and Jenkins for continuous integration and automated builds.

ID: DT 7	Name: Test Frameworks
Description	These are tools that are used for development and execution of tests.

3.10 Design Step 9: Validation and Testing

ID: DS 9	Name: Validation and Testing	Involved Roles: Software Architects, Developers, Testers, Control Engineers, Systems Engineers
Classification	Software Engineering	
Goal	• To check if the software components work according to the specification • To facilitate simulation as early as possible in order to reduce development costs	
Description	The software components are tested by the responsible roles to check if they are working as desired, i.e., according to the specified requirements.	
Sub-Activities	DS 9.1: Integrate with Controller and Environment DS 9.2: Model in the Loop DS 9.3: Rapid Control Prototyping DS 9.4: Software in the Loop DS 9.5: Hardware in the Loop DS 9.6: Software System Testing	
Uses	DC 17: Software Architecture Document DC 24: Component Tests DC 37: Control Model DC 38: Multi-domain Plant Model DC 39: Functional Mock-up Interface DC 40: Deployable Controller Software DC 41: Software Test Report Template	
Tools	DT 8: Simulation Tools DT 9: Code Generation Tools	
Provides	DC 42: Integrated System Simulation DC 43: Control Unit DC 44: Software Test Report DC 45: Software Package and Documentation	

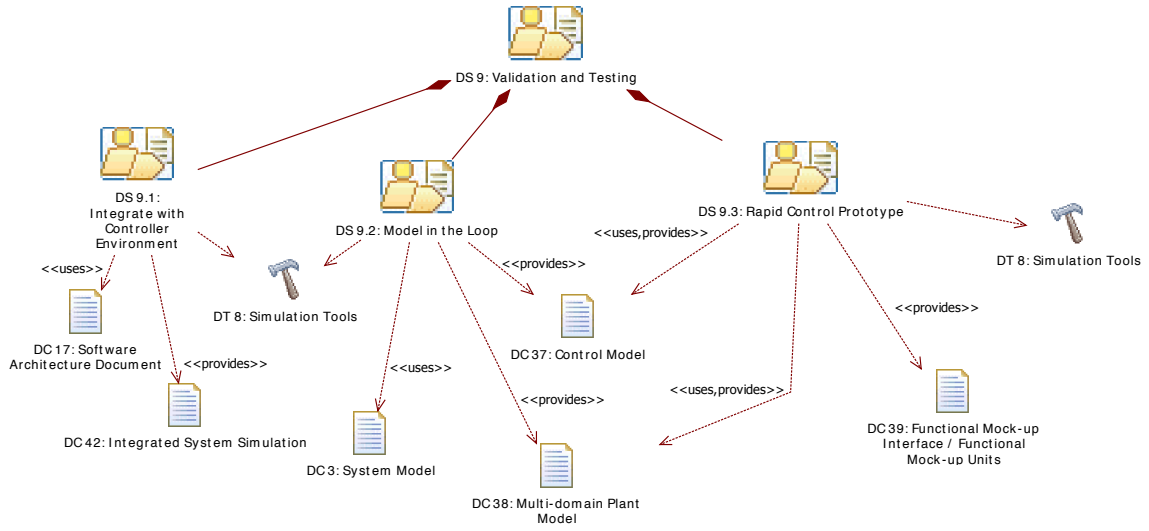

Figure 3.12: The steps, concepts, and tools used in design step DS9 (part 1/2).

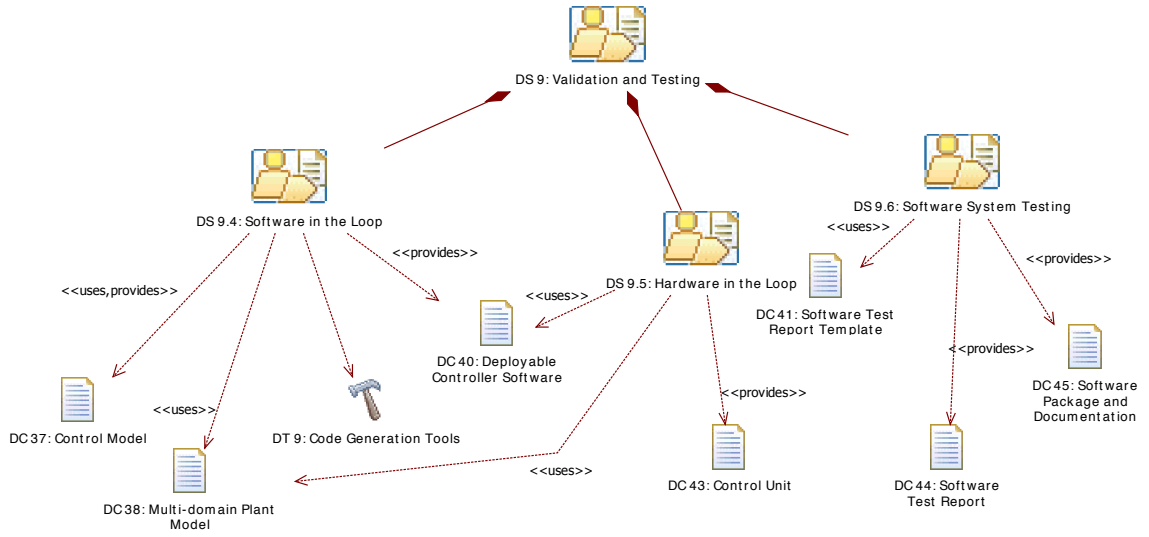

Figure 3.13: The steps, concepts, and tools used in design step DS9 (part 2/2).

3.10.1 Detailed Steps

ID: DS 9.1	Name: Integrate with Controller Environment	Involved Roles: Software Architects, Developers, Systems Engineers
Classification	Software Engineering	
Description	The model of the discrete behaviour is integrated with the feedback controllers of the system. Model transformations to tools like MATLAB Stateflow or Dymola allow an early integrated system simulation.	
Uses	DC 17: Software Architecture Document	
Tools	DT 8: Simulation Tools	
Provides	DC 42: Integrated System Simulation	

ID: DS 9.2	Name: Model in the Loop(MiL)	Involved Roles: Control Engineers
Classification	Control Engineering	
Description	In the system design phase, early control models and plant models are developed using tools like MATLAB/Simulink Stateflow and Dymola respectively. The goal is an early multi-domain simulation and validation of the overall system behaviour. At this early stage of development, the plant model is not necessarily very detailed.	
Uses	DC 3: System Model	
Tools	DT 8: Simulation Tools	
Provides	DC 37: (initial) Control Model DC 38: (initial) Multi-domain Plant Model	

ID: DS 9.3	Name: Rapid Control Prototyping (RCP)	Involved Roles: Control Engineers
Classification	Control Engineering	
Description	In the domain specific control engineering, the control model is refined (based on the results of the Model in the loop analysis). The refined control model is then validated against an early laboratory prototype or a real test bench. A configuration tool with a graphical user interface (GUI) and hardware interface is required to connect the control model with the prototype. Typically, tools like dSPACE ControlDesk or NI VeriStand are used to provide such a configuration GUI. The initial plant model is refined based on the measurements against the laboratory prototype.	
Uses	DC 37: (initial) Control Model DC 38: (initial) Multi-domain Plant Model	
Tools	DT 8: Simulation Tools	
Provides	DC 37: (refined) Control Model DC 38: (refined) Multi-domain Plant Model DC 39: Functional Mock-up Interface / Functional Mock-up Units	

ID: DS 9.4	Name: Software in the Loop (SiL)	Involved Roles: Control Engineers
Classification	Software Engineering	
Description	In the domain specific control engineering, software is generated in the desired target language from the control model (cf. RCP) which is tested against the refined plant model afterwards. The control model is further refined and optimized based on this validation step.	
Uses	DC 37: (refined) Control Model DC 38: (refined) Multi-domain Plant Model	
Tools	DT 9: Code Generation Tools	
Provides	DC 37: (refined) Control Model DC 40: Deployable Controller Software	

ID: DS 9.5	Name: Hardware in the Loop	Involved Roles: Control Engineer
Classification	Software Engineering	
Description	In the system integration phase (component testing), the control unit (hardware) together with the developed software (cf. SiL) is tested against the refined plant model.	
Uses	DC 38: (refined) Multi-domain Plant Model DC 40: Deployable Controller Software	
Provides	DC 43: Control Unit (including software)	

ID: DS 9.6	Name: Software System Testing	Involved Roles: Testers
Classification	Software Engineering	
Description	The software is tested to determine if everything is working as it should. The results of the tests are documented and the final software to be delivered is packaged.	
Uses	DC 24: Component Tests DC 41: Software Test Report Template	
Provides	DC 44: Software Test Report DC 45: Software Package and Documentation	

3.10.2 Design Concepts

ID: DC 37	Name: Control Model
Description	This is a model that specifies the flow of control between components in a system.

ID: DC 38	Name: Multi-domain Plant Model
Description	This models the physical environment of the System under development from the requirements. The model includes elements from different domains depending on the system being modelled.

ID: DC 39	Name: Functional Mockup Interface / Functional Mockup Units
Description	The Functional Mockup Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. In practice, the FMI implementation by a software modelling tool enables the creation of a simulation model that can be interconnected or the creation of a software library called FMU (Functional Mock-up Unit).

ID: DC 40	Name: Deployable Control Software
Description	This is the packaged integrated software that is ready to be deployed on a specific hardware.

ID: DC 41	Name: Software Test Report Template
Description	This is a template which test reports have to follow.

ID: DC 42	Name: Integrated System Simulation
Description	This is a model that simulates the behaviour of the real system under development.

ID: DC 43	Name: Control Unit
Description	This refers to a combination of the control software and the hardware, i.e., the control software deployed on the hardware

ID: DC 44	Name: Software Test Report
Description	This document includes the results of Software System Tests.

ID: DC 45	Name: Software Package and Documentation
Description	This is the final software product and its documentation.

3.10.3 Design Tools

ID: DT 8	Name: Simulation Tools
Description	These are tools that provide simulation platforms and facilitate the simulation of integrated and complex systems. They facilitate software testing without the use of actual hardware. Examples of these tools are MATLAB/Simulink[2], Modelica/Dymola[3], dSPACE ControlDesk[4] and NIVeristand[5].

[2]MATLAB/Simulink (`http://se.mathworks.com/products/simulink/`)
[3]Modelica/Dymola (`http://www.modelon.com/products/dymola/`)
[4]ControlDesk(`https://www.dspace.com/en/inc/home/products/sw/experimentandvisualization/controldesk.cfm`)
[5]NIVeristand (`http://www.ni.com/veristand/`)

ID: DT 9	Name: Code Generation Tools
Description	These are tools that can generate code from provided models.

3.11 Design Step 10: System Integration

ID: DS 10	Name: System Integration	Involved Roles: Developers, Systems Engineers
Classification	Software Engineering, Systems Engineering	
Goal	• Configure hardware-dependent software for particular (hardware-independently modelled) application software. • Parallelize (sequential) software and distribute it to the Hardware	
Description	In this step, the software is partitioned and packaged so that it can be deployed on hardware in an effective manner. In product line environments this step usually takes place after software component specification and variant configuration but before the implementation/code generation step.	
Sub-Activities	DS 10.1 : Create Executables DS 10.2 : Partitioning DS 10.3 : Task Creation DS 10.4 : Target Mapping	
Uses	DC 34: Source Code DC 46: Software Model	
Tools	DT 10: Compile-tool-chain DT 11: Partitioning Tools DT 12: Mapping Tools	
Provides	DC 30: Basic software configuration DC 47: Executable file DC 48: Partitioned software model DC 49: Constraint model DC 50: Software Model with Tasks DC 51: Mapping Model DC 52: OS Model DC 53: Stimulation Model DC 54: Property Constraints Model	

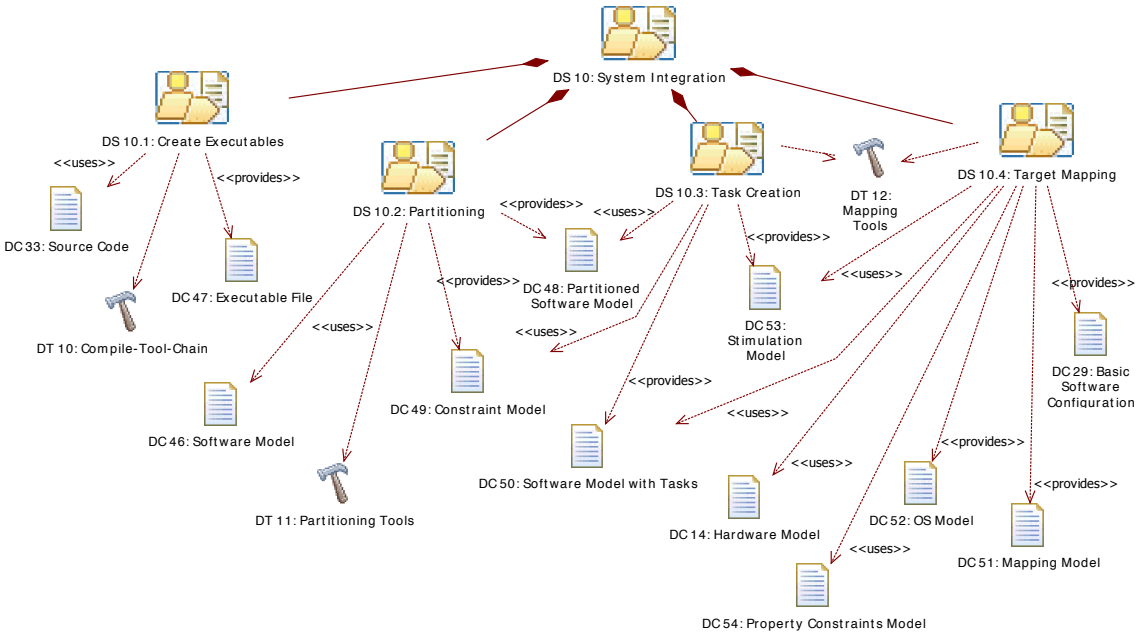

Figure 3.14: The steps, concepts, and tools used in design step DS10.

3.11.1 Detailed Steps

ID: DS 10.1	Name: Create Executables	Involved Roles: Developers
Classification	Software Engineering	
Description	Generate the executable for a target hardware to run software on the ECU.	
Uses	DC 33: Source Code	
Tools	DT 10: Compile-Tool-Chain	
Provides	DC 47: Executable File (*.elf)	

ID: DS 10.2	Name: Partitioning	Involved Roles: Developers
Classification	Software Engineering	
Description	Identify initial tasks by assessing graph structure and deriving possible partitions, that can be executed in parallel. Its configuration includes possibilities to: • group Runnables by their activation reference; • group independent set of Runnables (separate graphs); • define how cycles are decomposed in order to form DAGs (directed acyclic graphs). Firstly, the approach allows to identify a minimal number of edges, that have to be decomposed to eliminate cycles. Secondly, the approach identifies edges, that result in graphs with efficient parallelism potential in case they are selected for decomposition. • select from critical path partitioning (CPP) or earliest start scheduling partitioning (ESSP); • define whether a global critical path influences partition forming among all independent graphs and activation groups or if separate critical paths are formed for each independent graph and activation group for CPP; • define the number of partitions for ESSP.	
Uses	DC 46: Software Model	
Tools	DT 11: Partitioning Tools	
Provides	DC 48: Partitioned Software Model DC 49: Constraints Model	

ID: DS 10.3	Name: Task Creation	Involved Roles: Developers
Classification	Software Engineering	
Description	The Scope of this step lies in converting the Process Prototypes, which describe a preliminary version of tasks, into concrete tasks. These tasks agglomerate multiple Runnables into a larger group, which is allocated on the target platform. The Runnable Sequencing Constraints from the Constraints Model are used for identifying the Runnable order within the tasks. The results of this step are stored in an augmented Software Model. Since the groups of Runnables are reduced to a single common activation rate, a Stimulation model containing this activation is also created.	
Uses	DC 48: Partitioned Software Model DC 49: Constraints Model	
Tools	DT 12: Mapping Tools	
Provides	DC 50: Software Model with Tasks DC 53: Stimulation Model	

ID: DS 10.4	Name: Target Mapping	Involved Roles: Systems Engineers, Developers
Classification	Software Engineering, Systems Engineering	
Description	The scope of the mapping step lies in finding a valid and optimal distribution of software elements to hardware components. The mapping tool utilizes the models for software and hardware, as well as the tasks activation from the stimulation model, and calculates such a distribution. In order to calculate this distribution, several approaches are implemented, which can be selected and partially customized. Optionally, a Property Constraints Model may be included during the mapping process, which is used to narrow down the solution space, e.g. by enforcing a required attribute on a target core for a task. The results of this step are stored in a mapping model. Moreover, a preliminary OS model is generated, which contains a scheduler for each of the cores of the hardware platform.	
Uses	DC 50: Software Model with Tasks DC 14: Hardware Model DC 53: Stimulation Model DC 54: Property Constraints Model [opt]	
Tools	DT 12: Mapping Tools	
Provides	DC 29: Basic software configuration DC 51: Mapping Model DC 52: OS Model	

3.11.2 Design Concepts

ID: DC 47	Name: Executable File
Description	Executable file that can be deployed to an ECU to run software on target hardware.

ID: DC 48	Name: Partitioned Software Model
Description	The partitioned software model is an augmented version of the software model. It contains Process Prototypes, which agglomerate the runnables into groups.

ID: DC 49	Name: Constraints Model
Description	Constraints Models allow modelling different kinds of constraints, e.g., for describing runnable execution orders, affinities for the mapping of data to memories or timings. The partitioning tool uses the constraints model to store runnable dependencies in Runnable Sequencing Constraints and to derive a graph structure.

ID: DC 50	Name: Software Model with Tasks
Description	The Software Model with Tasks is an augmented version of the partitioned software model and contains task definitions which were generated from process types.

ID: DC 51	Name: Mapping Model
Description	Mapping Models describe the allocation of software elements to hardware components, e.g., which core is used to execute a specific executable (task), which memory is used to store data etc.

ID: DC 52	Name: OS Model
Description	OS Models describes the provided functionality of an operating system. They mainly provide a way to specify how access is given to certain system resources, e.g., schedulers or semaphores

ID: DC 53	Name: Stimulation Model
Description	Stimulation Models provide information about the activation times and rates of tasks.

ID: DC 54	Name: Property Constraints Model
Description	Property Constraints Models are used as an optional input and allow limiting the design space of a mapping by specifying required hardware characteristics to elements of the software.

3.11.3 Design Tools

ID: DT 10	Name: Compile-Tool Chain
Description	This is a collection of tools used for developing software and linking software to operating system and hardware.

ID: DT 11	Name: Partitioning Tools
Description	Partitioning tools identify possible partitions that can be executed in parallel. Therefore, it derives DAGs and forms partitions either based on the critical path or based on an earliest start schedule approach.

ID: DT 12	Name: Mapping Tools
Description	Mapping Tools allow creating tasks out of process prototypes. Furthermore, they implement several mapping strategies, which can be used to find an optimal distribution of software elements to hardware components.

3.12 Design Step 11: Handover

ID: DS 11	Name: Handover	Involved Roles: Project Managers, Developers, Customers, Sales Representatives
Classification	Software Engineering, Business	
Goal	To reach an agreement with the customer that the system has been implemented as specified by the agreed-upon requirements.	
Description	The acceptance tests should pass. A meeting is arranged with the customer where the test results are presented (alternatively the tests are run with the customer present). The customer signs a sign-off document.	
Sub-Activities	DS 11.1: Acceptance Testing DS 11.2: Delivery DS 11.3: Sign off	
Uses	DC 8: Acceptance Tests DC 12: Milestones	
Tools	DT 6: Collaboration tools	
Provides	DC 55: Acceptance Tests Results DC 56: Sign off Document	

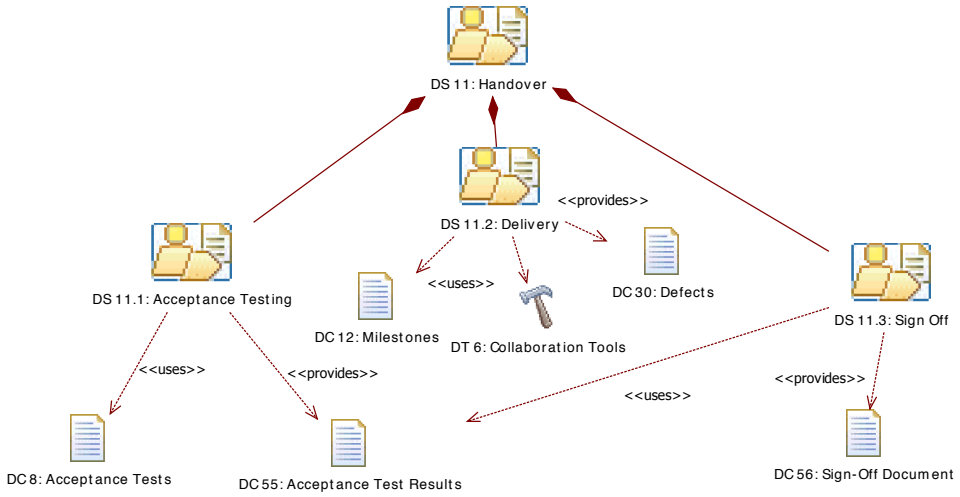

Figure 3.15: The steps, concepts, and tools used in design step DS11.

3.12.1 Detailed Steps

ID: DS 11.1	Name: Acceptance Testing	Involved Roles: Project Managers, Developers
Classification	Software Engineering	
Description	Acceptance tests which where developed according to the requirements are run. In order for the system to be delivered to the customer or to be considered done, all the tests should pass.	
Uses	DC 8: Acceptance Tests	
Provides	DC 55: Acceptance Tests Results	

ID: DS 11.2	Name: Delivery	Involved Roles: Project Managers, Developers, Customer
Classification	Software Engineering	
Description	The system (sources, tests) is delivered to the customer. A baseline is created (usually a classification in the revision control system). Acceptance tests are run and the results are documented (the tests do not have to pass until the final milestone). The customer uses the system and may report defects. The milestone is closed.	
Uses	DC 12: Milestones	
Tools	DT 6: Collaboration Tools	
Provides	DC 32: Defects	

ID: DS 11.3	Name: Sign-off	Involved Roles: Project Managers, Sales Representatives
Classification	Business	
Description	The acceptance tests should pass. A meeting is arranged with the customer where the test results are presented (alternatively the tests are run with the customer present). The customer signs a sign-off document.	
Uses	DC 53: Acceptance Test Results	
Provides	DC 56: Sign-off Document	

3.12.2 Design Concepts

ID: DC 55	Name: Acceptance Tests Results
Description	Documents the results of the acceptance tests.

3.12.3 Design Concepts

ID: DC 56	Name: Sign-off Document
Description	A document jointly signed by the customer and project manager, agreeing that the project has been implemented to specification, and also agreeing to close the project.

4 Exchange of Development Artefacts

During an embedded system development process, development artefacts must be exchanged for various reasons. For instance, development of different parts of the system is done by different developer teams or departments of a single company, each specialised for particular development steps. Different companies might cooperate in the development or parts of the development effort can be subcontracted to third parties. Even if software development was done by a single person, the application of different tools supporting dedicated development steps requires the exchange of development artefacts between those tools. In the context of multi- and many-core software development, this exchange gets even more important because pieces of software delivered by different suppliers must be deployed to a single hardware platform.

Nevertheless, the various steps within development of embedded multi- and many-core software should be connected to a continuous design flow including traceability. For this purpose, this chapter will cover requirements on the exchange of design concepts as well as use cases for the exchange of development artefacts. In addition, an overview of the exchange of artefacts between the design steps outlined in Chapter 3 will be given. Concrete examples of the exchange of artefacts in the context of AUTOSAR illustrate how it is handled in practice.

4.1 Requirements on the Exchange of the Design Concepts

The design concepts described in Chapter 3 can be categorised into documents that consist mostly of text, models that are usually described in a formal modelling language, and lists that follow a structured form but use no formal language. The requirements for the exchange of these three categories differ slightly, since they can be stored and transmitted in different ways. Generally, the exchange of artefacts requires *syntactic compatibility*, i.e., the file format must be readable by the other party, and *semantic compatibility*, i.e., the information stored in the file must be understandable by the other party.

Text documents. Text documents, e.g., review documents such as DC 18: Software Architecture Review Protocol Document, often have limited structure and are formulated in free text. This means that their syntactic compatibility can be ensured by storing them in a commonly accepted file format (such as Microsoft Word, OpenDocument, or Plain Text) and the semantic compatibility is not strictly enforced but usually implicitly given by the use of a language all parties understand and by the subject matter. These documents are not intended for consumption by automated tools and are generated either manually or automatically from a more structured document.

Models. Models capture highly structured information in a precisely defined format. They are used to capture most technical information, e.g., DC 3: System Model or DC 14: Hardware Model. Each element in a model is assigned a concrete semantic meaning by making it an instance of a meta-model. The most commonly used meta-models are well-known industry

standards (e.g., SysML or UML) and the AMALTHEA platform defines a number of meta-models for various purposes as well. Semantic compatibility is therefore given by the adherence to these meta-models and can be established by exchanging the meta-model along with the models. This enables the use of automated techniques to, e.g., transform a model. It must be noted, however, that on a higher level, semantic information encoded in the model must still be interpreted by a human. Syntactic compatibility is usually ensured through the use of interchange formats. While there is a multitude of modelling tools available (e.g., MagicDraw, StarUML, Papyrus), each with its own file format, the XML Metadata Interchange (XMI) format has been established as a de facto model exchange format.

Semi-structured lists. Such documents, e.g., DC 12: Milestones or DC 32: Defects, have more structure than plain text documents since they are usually in the form of tables or matrices and the content can therefore be assigned to specific denominators. This also implies a certain level of semantics, but since the entries themselves are still formulated in natural language, assigning meaning to the entries still requires a human reader. As for the file format and therefore the syntactic compatibility, the same remarks as for text documents apply.

The use of version control systems (VCS) such as Git or Subversion poses a challenge when binary formats such as OpenDocument or Microsoft Word are used. Since VCS are not capable of computing the differences between two versions of the same binary file (again, an issue of both syntactical and semantic compatibility), features such as comparison (diffing) and automated merging are not available. Conflicts must be painstakingly resolved manually by the stakeholders.

Using online collaboration tools can in many cases make it easier to exchange information between developers and even between sub-units in a company or between companies. Instead of storing and exchanging files, potentially through a VCS with limited effect, these tools allow viewing and creating artefacts online in a unified platform. Tools like Trac, Jira, Trello and others thus alleviate issues of syntactic compatibility. However, the use of these tools requires that all parties are willing and able to interact with these tools both from a technical and a policy perspective. Offline availability and mobile usage may be limited.

It must be noted that a specific artefact can be in either of the three forms mentioned above. A requirements document such as DC 4: System Requirements Specification Document can, e.g., be a simple text document, be a structured list expressed as a table, or can be a clearly defined model based, e.g., on the UML meta-model. While the latter possibility incurs a higher workload in the initial definition of the artefact, it might have benefits if the document is re-used and exchanged later on. Which concrete form each of the artefacts in the development process takes is a decision that must be made by the project stakeholders while the process is tailored to the project. These decisions are usually influenced by organisational rules, industry best practices, the expected complexity of the project, and the involved parties.

4.2 Use Cases for the Exchange of Artefacts

Artefacts are exchanged in every development process that includes more than one person or more than one tool. Two general cases can be distinguished: exchange within the same development team and exchange with people from outside the development team. The main difference is that a development team usually has an implicit notion of the semantics of the

artefacts, thus providing semantic compatibility for artefacts provided and used by that team. Should such compatibility not exist in specific cases, the short communication routes within the team allow its establishment in an ad-hoc fashion if necessary. On the other hand, when exchanging artefacts with outsiders, regardless of whether they are part of the same company or a different company, this implicit semantic understanding may not exist. In addition, since the tool chains may differ, even syntactic understanding becomes problematic.

The following paragraphs outline four use cases for the exchange of artefacts where the first two belong to the category where implicit semantic compatibility is assumed and the other two to the category where this can not be taken for granted. The use cases are accompanied by two figures: Figure 4.1 shows which artefacts are exchanged between the design steps outlined in Chapter 3 in general and Figure 4.2 shows which artefacts are potentially exchanged between organisations.

Exchange of artefacts between tools The descriptions in Chapter 3 show that each design step consumes and produces one or more design artefacts. The production and consumption of the artefacts is facilitated by the use of various tools. For an artefact from one design step using a particular tool to be used in another design step using another tool, the format of the artefact needs to be readable by the two tools, i.e., output from one tool needs to be recognised as input to the other tool (syntactic compatibility, see above).

For tools that are not compatible with each other, interfacing mechanisms can be applied to facilitate the exchange of information between them. For example Open Services for Lifecycle Collaboration (OSLC) is an open standard that provides a way to connect two tools together so that one of the tools can access resources of the other tool. For some specific elements, universally accepted standards exist (e.g., the UML meta-model or the ReqIF format for requirements) that can facilitate such an exchange.

Exchange of artefacts between developers on the same team Development teams consists of a number of developers who work on achieving certain milestones in the development of a system. Although each developer can work on a task alone, it is often the case that the result of a task needs to be used by other developers in the team as all of them are working on developing the same end product. This exchange usually pertains to artefacts within the same tools but great importance must be given to the correct version of the exchanged artefact. This is achieved by using VCS such as Git or Subversion and applying a defined process for committing and retrieving changes.

Exchange of artefacts between sub-units in the same company Sometimes artefacts need to be exchanged between different development teams or different departments in the same company for further development or for use as input to other design steps. These concepts can be exchanged between the same tools or different tools. In case there are different tools, the same remarks discussed in the first use case apply.

Exchange of artefacts between different companies When the development or part of the development of a system is subcontracted to a third party company, the OEM needs to exchange development artefacts with the third party company. The goal is usually to provide information about a system as well as requirements to the involved 3rd party. The 3rd party can then give additional input to the exchanged artefact and then send it back. The contents of the exchanged artefact can be obscured, if there is the need for

Intellectual Property (IP) protection. For instance this can be done by using randomly generated names for parts of the artefact.

In order for the exchanged artefact to be useful to the 3rd party, other information may be required as well, e.g., information related to customer requirements. Therefore there is a need to export and exchange such information as well, together with the traceability information between the artefacts.

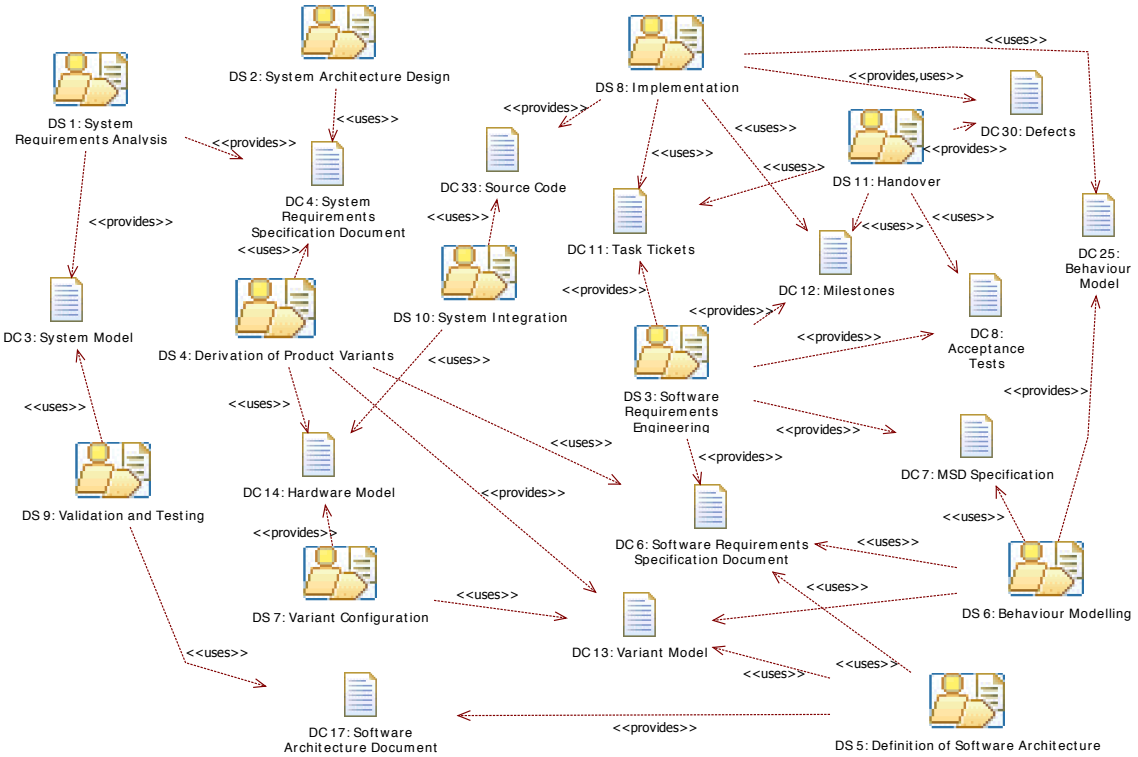

Figure 4.1: Exchange of artefacts between design steps. Since these design steps can be performed by different division within the same company or by different companies altogether, the exchanged artefacts pose special requirements on form, traceability, and format.

4.3 Examples for Exchanging Development Artefacts

This section provides some examples how exchange of development artefacts is currently handled with a special focus on the standardisation provided by AUTOSAR.

4.3.1 AUTOSAR: From Software Architecture to System Integration

As described in Chapter 2, AUTOSAR specifies a layered system architecture aiming at enabling hardware-independent application software development. However, deployment of hardware-independent application software requires a system integration step to enable software execution on a dedicated hardware platform. For this purpose, AUTOSAR specifies hardware-dependent

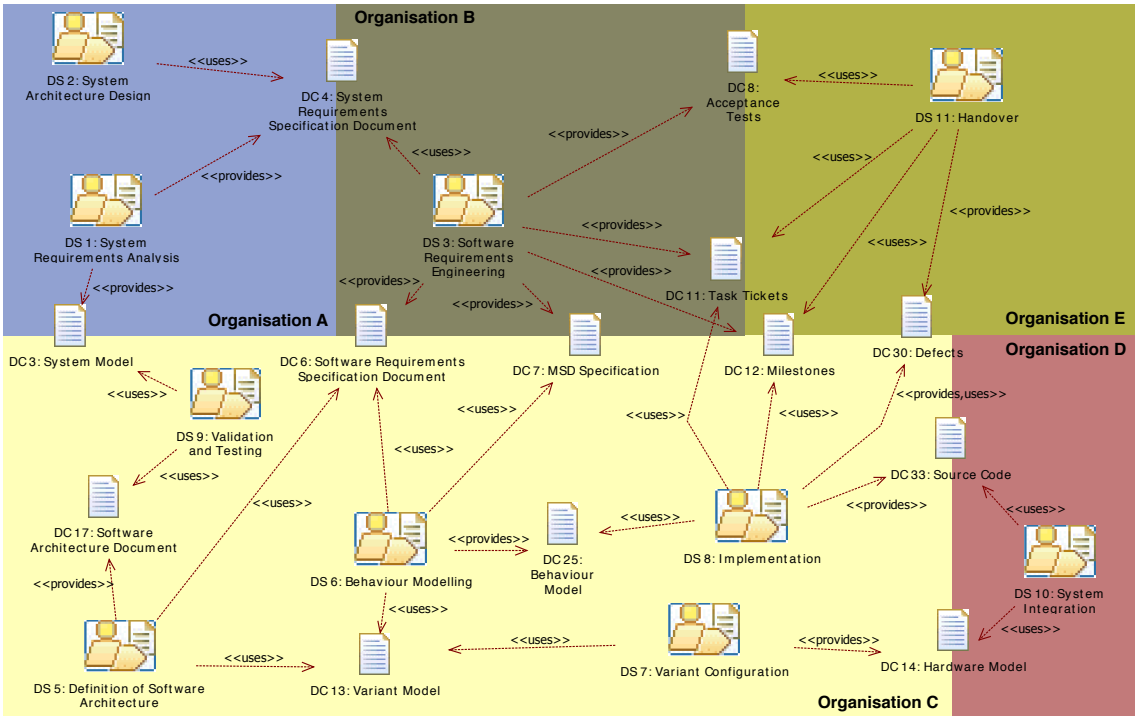

Figure 4.2: Exchange of artefacts between different organisations. An organisation can be a development team, a department, or a company. In the first two cases, the exchange happens within the same company, potentially simplifying it since similar processes, collaboration tools, and communication patterns are used. In the last case, documents are delivered and very little communication apart from that can be assumed, thus representing a typical outsourcing scenario. The most important exchanged artefacts are shown here. Central artefacts are the outcomes of the requirements engineering step as well as code and documentation for the handover of the finished product. Please note that the organisations do not necessarily have to be distinct.

basic software (BSW) that is connected to hardware-independent application software by the AUTOSAR runtime environment (RTE) (Figure 4.3).

This example for exchanging development artefacts demonstrates how the transition from design step 5 “Definition of Software Architecture” to design step 10 “System Integration” (cf. Chapter 3) can be performed by means of AUTOSAR.

AUTOSAR provides different views on the software architecture, each providing a different level of detail. Here, we focus on the Virtual Function Bus (VFB) level where application software is modelled by software components (SWC). Such a SWC encapsulates one or more Runnables, each representing some functionality. Runnables in turn are the smallest schedulable entity covered by AUTOSAR. Furthermore, a SWC has ports to model required input as well as provided output data flow. Ports are therefore associated with port interfaces that describe data, e.g., by means of data type. Another property of ports is their communication mechanism. AUTOSAR allows to specify Sender-Receiver or Server-Client communication, so ports are characterized as sender, receiver, server, or client port. At VFB level, system architects can instantiate various SWCs and connect them via their ports. However, ports cannot arbitrarily

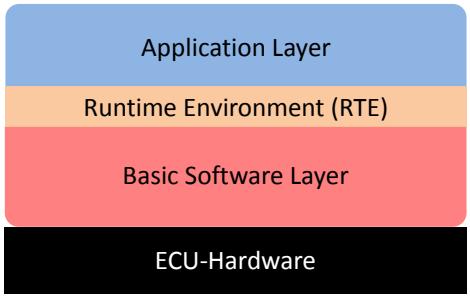

Figure 4.3: AUTOSAR software layers (cf. [7]).

be connected but have to be compatible w.r.t. the addressed communication mechanism and their port interfaces.

The textual, domain specific language "Software Component Language" (SWC Language) – specified by ARText that is based on Xtext [15] and provided by ARTOP [4] – enables textual definition of AUTOSAR-compliant SWCs. Amongst others, it provides means to define ports, port interfaces, internal behaviour with Runnables and VFB-Events, and even compositions of SWCs. SWC Language stores data in a software component description file (`*.swcd`). But ARText provides also means to export data to an AUTOSAR XML file (`*.arxml`). Hence, definition of software architecture covering instantiations of SWCs and their data dependencies given by connections of their ports can also be exported to an AUTOSAR XML file.

The tool "Arctic Studio", an Eclipse-based, commercial development environment for Arctic Core AUTOSAR platform provided by ArcCore [3], provides this SWC Language to define AUTOSAR-compliant software architecture. For system integration, Arctic Studio provides configuration of AUTOSAR BSW to integrate SWCs on a target platform. Import of system architecture information is enabled by importing AUTOSAR XML files.

This way, AUTOSAR XML files provide a means to pass development artefacts resulting from design step 5 "Definition of Software Architecture" to design step 10 "System Integration". In design step 10, imported information regarding software architecture are used to appropriately configure BSW and generate RTE.

Figure 4.4 shows an example of how this exchange of artefacts is enabled by means of AUTOSAR. First, a system architect defines the software architecture by a set of software components, each stored in a separate **.swcd*-file. The collection of SWC description files is exported to a single AUTOSAR xml-file (**.arxml*). This file, in turn, is imported by ArcticStudio and provides information like communication between various SWCs. This data are required during system integration, e.g., to configure and generate AUTOSAR runtime environment (RTE) and configure network communication.

4.3.2 Exchange of Artefacts between Tools: From Software Architecture to Behaviour Modelling to Code Generation

This example demonstrates how design artefacts are exchanged between various tools. The example assumes that a company is using model-driven software development. The example is illustrated using Figure 4.5 which shows a transfer of design artefacts from a component modelling tool to a behaviour modelling tool and further to a code generation tool.

The architecture of the system which describes the software components of the SUD is mod-

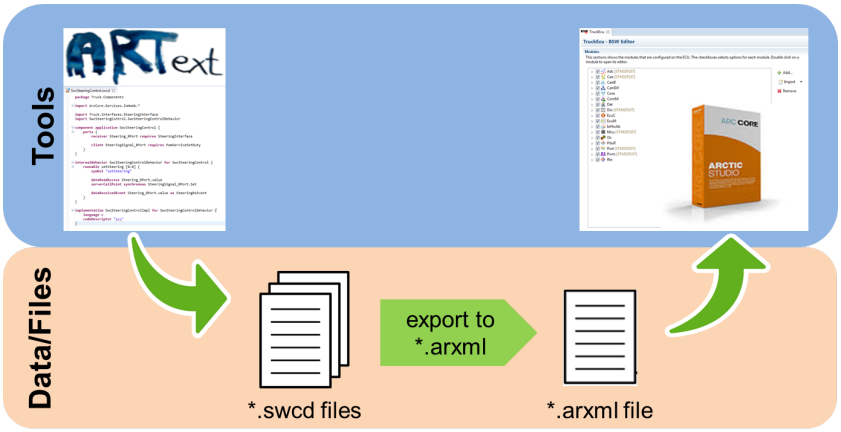

Figure 4.4: From software architecture modelling to system integration: Software architecture is modelled by means of ARText-based SWC Language, exported to AUTOSAR xml-file, and imported by ArcticStudio for system integration.

elled using a specification tool such as Vehicle System Architect (VSA) – step 1 in Figure 4.5. This can then be exported in the AUTOSAR standard exchange format (.arxml) – step 2 in Figure 4.5. The .arxml file is then imported to a behaviour modelling tool such as Simulink – step 3 in Figure 4.5 – to allow the behaviour of the components to be modelled. When modelling the behaviour, the architecture (software component model) is also refined until a final working version that can be used to generate code is obtained. Early versions of the behaviour model are tested by simulations using Simulink – step 4 in Figure 4.5 – and later the final versions are tested in the actual vehicle. The final behaviour model is used to generate code for the software and the amended architecture is exported back to the specification tool (in this case VSA) for the changes to be merged to the original architecture – step 5 in Figure 4.5.

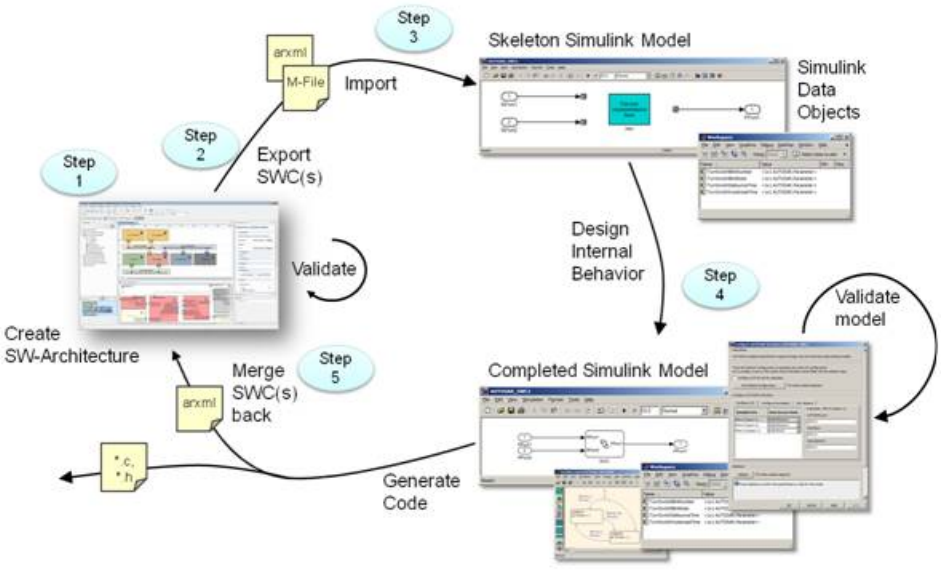

Figure 4.5: Exchange of Artefacts between Tools [19].

5 Preliminary Compatibility Analysis for Design Steps with ISO 26262

Safety is a critical aspect in the automotive industry. Due to this fact and since the identified design steps address suppliers for the automotive industry and academic partners with experience from such suppliers, safety plays an important role in this document. This chapter gives an evaluation of the design steps described in Chapter 3 in the context of the ISO 26262 standard which has been introduced in Section 2.4 of Chapter 2. It is based on the review of the design steps conducted by OFFIS which is one of the project partners that has expertise with the ISO 26262 standard. The aim is to show which of the ISO 26262 phases the design steps cover and which ones are not covered at the moment. A phase or sub-phase in the ISO 26262 standard is covered by a design step when the activities in the design step are similar or correspond to the activities in the phase or sub-phase. This section also points out the weakness in the design steps and proposes ways of improvement to increase compatibility with the ISO 26262 standard.

The ISO 26262 standard has 10 main phases divided into several sub-phases as shown in Figure 5.1. The design steps discussed in this document cover the sub-phases which are marked by the green borders on the figure. Sub-phases not marked explicitly are not covered by the identified design steps at the moment and will be subject to further investigation.

The design steps in this document have a distinction between the system engineering level and the software engineering level that is marked by the tags in the tables describing the steps. This distinction corresponds with the distinction made in the ISO 26262 phases Phase 4: Product development at the system level and Phase 6: Product development at the software level. The rest of the phases are out of scope for this document.

In phase 4, the design step DS 1: System Requirements Analysis corresponds to the sub-phase 4-5: Initiation of product development at the system level. The design step DS 2: System Architecture Design corresponds to the sub-phase 4-7: System design. The sub-phase 4-8 which is item integration and testing corresponds to the design step DS 10: System Integration while the sub-phase 4-11 which is release for production corresponds to DS 11: Handover. For phase 6, the design steps' correspondence to the sub-phases are shown in the table below.

Even though the identified design steps cover some aspects of verification and validation, the steps do not explicitly mention the verification and validation of safety requirements. This is why the sub-phases 4-6, 4-9, 4-10 and 6-11 are not covered by any design steps in particular. This is a weakness in the design steps because they are not concrete enough to reveal the safety related aspects.

In order for the steps to be completely compatible with the standard, more concrete guidelines need to be in place on how to use the design steps. For instance the ISO 26262 standard explicitly requires that the requirement template should have an attribute for a unique ID and a distinction of whether a requirement is related to safety or not. If they are, an additional attribute called ASIL (Automotive Safety Integrity Level) is required. Subsystems have to inherit the safety-status and the ASIL from their parental element. Therefore their structure

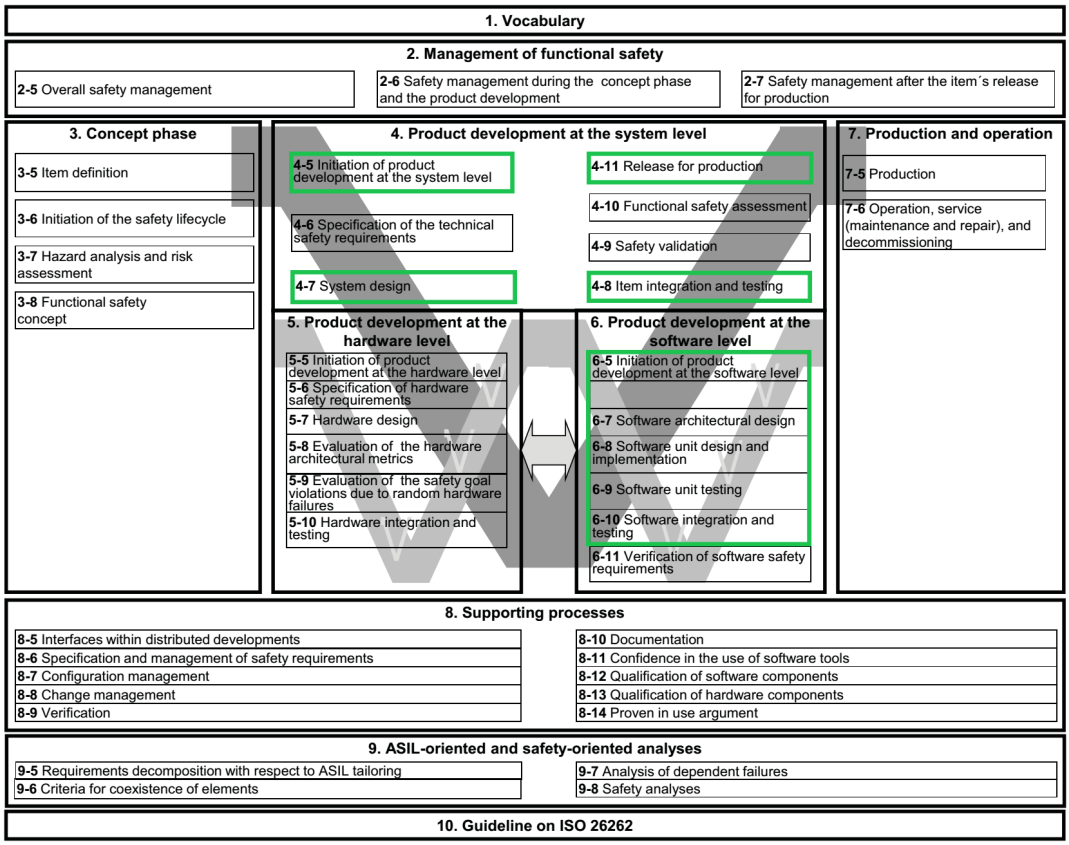

Figure 5.1: Overview of the different parts of ISO 26262 covered by the design steps

must be comprehensible. Also the standard requires that every verification or validation step has a stated goal on whether it checks for correctness, completeness or consistency. The standard also emphasises traceability of safety related artefacts from requirements down to their implementation and tests.

These more concrete guidelines on how the steps should be used in order to be compatible with the ISO 26262 standard will be created based on further data collection and evaluation from the project partners. The guideline will be part of the last deliverable for the project, a design handbook that will illustrate the design flow supported by the techniques developed in AMALTHEA4public and the AMALTHEA4public tool chain.

Sub-phase	**Design step(s)**
6-5 Initiation of product development at the software level	DS 3: Software Requirements Engineering
6-7 Software architectural design	DS 4: Derivation of Product Variants DS 5: Definition of Software Architecture DS 6: Behaviour Modelling
6-8 Software unit design and implementation	DS 7: Variant Configuration DS 8: Implementation
6-9 Software integration and testing	DS 9: Validation and Testing

Table 5.1: ISO 26262 sub-phases of phase 6 and corresponding design steps.

6 Conclusion

This deliverable provides an overview of the design steps currently in use in the development of embedded software with a special focus on the automotive domain. It illustrates which design goals and tools are involved, which artefacts are created and exchanged, and how the identified steps are compatible with ISO26262.

The work presented here gave important insights into the workflows used by the project partners in AMALTHEA4public and will help us to achieve the next two goals in the work package in tasks 1.4 and 1.7: definition of a traceability concept and design of an enhanced design flow. It also allows to map the artefacts to the models the AMALTHEA tool chain provides and thus to identify which parts of the design steps can not be accomplished by the current tool chain.

With regard to the traceability concept, the overview of identified artefacts and how they are exchanged between design steps, development teams, or companies, allows us to refine the requirements on traceability already collected. We now have a complete view of the models that are used and how they are connected through the design steps that allows us to look at the specific traceability needs for each kind of artefact. In addition, the identified patterns of exchange allow us to analyse which kind of traceability information must be preserved when an exchange takes place and which technical infrastructure is necessary to ensure this. This collaborative view on traceability is novel and will be a focus of investigation in the next year.

Finally, the design of an enhanced design flow that will incorporate the techniques and approaches developed in AMALTHEA4public will be greatly simplified due to the analysis presented in this document. With an understanding of the existing workflows and design steps, we can now work in close collaboration with our project partners in order to seamlessly integrate new aspects such as safety, the already mentioned traceability, as well as improvements in the product line approaches.

Bibliography

[1] IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. In: *IEEE Std 1471-2000* (2000), S. i–23

[2] Apel, Sven ; Kästner, C.: An overview of feature-oriented software development. In: *Journal of Object Technology (JOT)* 8 (2009), Nr. 5, S. 49–84

[3] ArcCore AB: *Homepage of ArcCore*. http://www.arccore.com/. 2015

[4] Artop - AUTOSAR Tool Platform User Group: *Homepage of Artop*. http://www.artop.org/. 2015

[5] AUTOSAR GbR: *Feature Model Exchange Format*. http://www.autosar.org/fileadmin/files/releases/4-2/methodology-templates/templates/standard/AUTOSAR_TPS_FeatureModelExchangeFormat.pdf. 2014

[6] AUTOSAR GbR: *Generic Structure Template*. http://www.autosar.org/fileadmin/files/releases/4-2/methodology-templates/templates/standard/AUTOSAR_TPS_GenericStructureTemplate.pdf. 2014

[7] AUTOSAR GbR: *Homepage of AUTOSAR*. http://www.autosar.org/. 2014

[8] AUTOSAR GbR: *Methodology*. http://www.autosar.org/fileadmin/files/releases/4-2/methodology-templates/methodology/auxiliary/AUTOSAR_TR_Methodology.pdf. 2014

[9] AUTOSAR GbR: *Virtual Functional Bus*. http://www.autosar.org/fileadmin/files/releases/4-2/main/auxiliary/AUTOSAR_EXP_VFB.pdf. 2014

[10] Böckle, Günter ; Pohl, Klaus ; Linden, Frank van der: *Software Product Line Engineering*. Springer Berlin Heidelberg, 2005. – ISBN 978-3-540-24372-4

[11] Braun, Peter ; Broy, Manfred ; Houdek, Frank ; Kirchmayr, Matthias ; Müller, Mark ; Penzenstadler, Birgit ; Pohl, Klaus ; Weyer, Thorsten: Guiding requirements engineering for software-intensive embedded systems in the automotive industry. In: *Computer Science - Research and Development* 29 (2014), Nr. 1, S. 21–43. – URL http://dx.doi.org/10.1007/s00450-010-0136-y. – ISSN 1865-2034

[12] Broy, Manfred ; Rausch, Andreas: Das neue V-Modell XT. In: *Informatik-Spektrum* 28 (2005), Nr. 3, S. 220–229. – URL http://dx.doi.org/10.1007/s00287-005-0488-z. – ISSN 0170-6012

[13] Bürdek, Johannes ; Lity, Sascha ; Lochau, Malte ; Berens, Markus ; Goltz, Ursula ; Schürr, Andy: Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints. In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, 2013 (VaMoS '14), S. 16:1–16:8

[14] Clements, Paul ; Northrop, Linda: *Software Product Lines: Practices and Patterns*. SEI Series. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2001. – 563 S. – ISBN 978-0-201-70332-0

[15] contributors, Various open-source: *Xtext*. July 2015. – URL `http://www.eclipse.org/Xtext/`

[16] Department of Transportation, Office of Operations: Systems engineering for intelligent transportation systems / Federal Highway Administration & Federal Transit Administration. URL `http://ops.fhwa.dot.gov/publications/seitsguide/`, January 2007 (FHWA-HOP-07-069). – Forschungsbericht

[17] ISO: *ISO 26262 - Road vehicles — Functional safety — Part 1 Vocabulary*. Juli 2009

[18] Kang, K. C. ; Cohen, S. ; Hess, J. ; Novak, W. ; Peterson, A.: Feature-Oriented Domain Analysis (FODA), Feasibility Study / Software Engineering Institute. Carnegie Mellon University, 1990 (CMU/SEI-90-TR-21). – Forschungsbericht

[19] Michael Seibt, Guido Sandman: *Stepping through the AUTOSAR round trip*. `http://www.electronicsnews.com.au/Features/Stepping-through-the-AUTOSAR-round-trip`. 2015

[20] OMG: *Software & Systems Process Engineering Meta-Model Specification Version 2.0*. : Object Management Group (Veranst.), April 2008. – URL `http://www.omg.org/spec/SPEM/2.0/`

[21] Osborne, Leon ; Brummond, Jeffrey ; Hart, Robert D. ; Zarean, Mohsen ; Conger, Steven M.: Clarus: Concept of operations / US Department of Transportation – Federal Highway Administration. URL `http://ntl.bts.gov/lib/jpodocs/repts_te/14158.htm`, October 2005 (FHWA-JPO-05-072). – Forschungsbericht

[22] Pohl, Klaus ; Hönninger, Harald ; Achatz, Reinhold ; Broy, Manfred: *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*. Springer Publishing Company, Incorporated, 2012. – ISBN 3642346138, 9783642346132

[23] SPES 2020 project consortium: *Homepage of the SPES 2020 project*. `http://www.spes2020.de/`. 2015

[24] SPES XT project consortium: *Presentations on highlights and use cases of the SPES XT project*. `http://spes2020.informatik.tu-muenchen.de/abschlussevent_xt.html`. 2015